A distributed API for coordinating AbC programs

Citation for the original published paper (version of record):
Alrahman, Y., Garbi, G. (2020). A distributed API for coordinating AbC programs. International Journal on Software Tools for Technology Transfer, 22(4): 477-496.
http://dx.doi.org/10.1007/s10009-020-00553-4

International Journal on Software Tools for Technology Transfer (2020) 22:477–496
https://doi.org/10.1007/s10009-020-00553-4

FOUNDATION FOR MASTERING CHANGE

Special Section REoCAS

A distributed API for coordinating *AbC* programs

Yehia Abd Alrahman[1] · Giulio Garbi[2]

Published online: 28 February 2020

Abstract
Collective adaptive systems exhibit a particular notion of interaction where environmental conditions largely influence interactions. Previously, we proposed a calculus, named *AbC*, to model and reason about CAS. The calculus proved to be effective by naturally modelling essential CAS features. However, the question on the tradeoff between its expressiveness and its efficiency, when implemented to program CAS applications, is to be answered. In this article, we propose an efficient and distributed coordination infrastructure for *AbC*. We prove its correctness, and we evaluate its performance. The main novelty of our approach is that *AbC* components are infrastructure agnostic. Thus the code of a component does not specify how messages are routed in the infrastructure but rather what properties a target component must satisfy. We also developed a Go API, named *GoAt*, and an Eclipse plugin to program in a high-level syntax which can be automatically used to generate matching Go code. We showcase our development through a non-trivial case study.

Keywords Attribute-based interaction · Semantics · Process calculi · Programming API

1 Introduction

Collective adaptive systems (CAS) [20] consist of a large number of components that interact anonymously, based on their properties and on contextual data, and combine their behaviours to achieve global goals. The boundaries of CAS are fluid, and components may enter or leave the system at any time. Components may also adapt to their environmental conditions.

Most of the current communication models cannot naturally model highly adaptive and loosely coupled systems with fluid boundaries like CAS. They actually suffer from limitations due to: lack of knowledge representation, e.g. π-calculus [28] or rigid communication interfaces, e.g. CBS [31]. To mitigate the shortcomings of the current communication paradigms when dealing with CAS, we have proposed a kernel calculus, named *AbC* [3,9], to program CAS interactions. The idea is to permit the construction of formally verifiable CAS systems by relying on a minimal set of interaction primitives. Such primitives provide a language-based approach to programming and thus can be used to implement different features and mechanisms. Clearly, this is more desirable when compared to self-organising algorithms [12,15] that are developed for specific features.

AbC's primitives are attribute-based [1,4] and abstract from the underlying coordination infrastructure (i.e. they are infrastructure agnostic). They rely on *anonymous* multicast communication where components interact based on mutual interests. Message transmission is non-blocking, while reception is not. Each component has a set of attributes to represent its run-time status. Communication actions (both send and receive) are decorated with predicates over attributes that partners have to satisfy to make the interaction possible. The interaction predicates are also parametrised with local attribute values, and when values change, the interaction groups do implicitly change, introducing opportunistic interactions.

Basing the interaction on run-time attribute values is indeed a nice idea, but it needs to be supported by a middle-

We want to thank Rocco De Nicola and Michele Loreti for early discussions about the conference version of this article. Yehia Abd Alrahman is funded by the ERC consolidator grant D-SynMA under the European Union's Horizon 2020 research and innovation programme (Grant No 772459).

✉ Yehia Abd Alrahman
yehia.abd.alrahman@gu.se

[1] University of Gothenburg, Chalmers University of Technology, Gothenburg, Sweden

[2] IMT School for Advanced Studies Lucca, Lucca, Italy

ware that provides efficient ways for distributing messages, checking attribute values, and updating them. A typical approach is to rely on a centralised broker that keeps track of all components, intercepts every message and forwards it to registered components. It is then the responsibility of each component to decide whether to receive or discard the message. This is the approach used in the Java-based implementation [2] of *AbC*. Clearly, any centralised solution may not scale with CAS dynamics and thus becomes a bottleneck for performance. A distributed approach is definitely preferable for large systems. However, distributed coordination infrastructures for managing the interaction are still scarce and/or inefficient [34]. Also the correctness of their overall behaviour is often not obvious.

One solution is to rely on existing protocols for total-order broadcast to handle message exchange. However, these protocols are mostly centralised [18] or rely on consensus [34]. Centralised solutions have scalability and efficiency issues, while consensus-based ones are not only inefficient but also impossible in asynchronous systems where components might fail [21]. Consensus algorithms also assume that components know each other and can agree on a specific ordering. However, this contradicts the main design principles of the *AbC* calculus where anonymity and openendedness are crucial factors. Since *AbC* components are *agnostic* to the infrastructure, they cannot participate in establishing a total ordering. Thus, we need an infrastructure that guarantees total ordering seamlessly.

In this article, we present a theoretical foundation of a distributed coordination infrastructure for message exchange and prove its correctness with respect to the original semantics of *AbC* [9]. We also provide an actual implementation of this infrastructure and evaluate its performance by means of stochastic simulation. The infrastructure is implemented in Google Go because we believe that Go is more appropriate to deal with CAS due to its clean concurrency model. In essence, we provide an *attribute-based API* for Go, named $\mathcal{GoAt}$, with the goal of using the *AbC* primitives to program the interaction of CAS applications directly in Go. The actual implementation of $\mathcal{GoAt}$ fully relies on the formal semantics of *AbC* and is parametric with respect to the infrastructure that mediates interactions. We provide a one-to-one correspondence between the *AbC* primitives and the programming constructs of $\mathcal{GoAt}$. We also provide an Eclipse plugin for $\mathcal{GoAt}$ to permit programming in a high-level syntax which can be analysed via formal methods by relying on the operational semantics of *AbC*. This article is an extended and a more refined version of the works presented in [6,7]. Here we enhance the presentation, fix imprecisions, and provide the proof details of the correctness of the proposed infrastructure.

The rest of this article is structured as follows: In Sect. 2, we briefly review the *AbC* calculus. In Sect. 3, we give a full formal account of a distributed coordination infrastructure for *AbC* and its correctness. In Sect. 4, we present the $\mathcal{GoAt}$ API and show how to program a distributed graph colouring algorithm in $\mathcal{GoAt}$ and we briefly comment on the Eclipse plugin. In Sect. 5, we provide a detailed performance evaluation. Finally, in Sects. 6 and 7 we conclude the article and survey-related works.

2 The *AbC* calculus

In this section, we briefly introduce the *AbC* calculus by means of a running example. We give *an intuition* of how to model a distributed variant of the well-known *Graph Colouring Problem* [24] using *AbC* constructs. We render the problem as a typical CAS scenario where a collective of agents, executing the same code, collaborate to achieve a system-level goal without any centralised control. The presentation is intended to be intuitive, and full details of the syntax and the semantics of *AbC* can be found in [3,9]. The example will be presented thoroughly in Sect. 4.3.

The problem consists of assigning a *colour* (an integer) to each vertex in a graph while avoiding that two neighbours get the same colour. The algorithm consists of a sequence of rounds for colour selection. At the end of each round, at least one vertex is assigned a colour. A vertex, with identity id, uses messages of the form $(\text{“}try\text{"}, c, r)$ to inform its neighbours that at round r it wants to select colour c and messages of the form $(\text{“}done\text{"}, c, r)$ to communicate that it has chosen colour c at the end of round r. At the beginning of a round, each vertex selects a colour and sends a *try*-message to all of its neighbours N. A vertex also collects *try*-messages from its neighbours. The selected colour is assigned to a vertex only if it has the greatest id among those that have selected the same colour in that round. After the assignment, a *done*-message (associated with the current round) is sent to neighbours.

***AbC* Syntax.** A component, $\Gamma :_I P$, is a process P associated with an *attribute environment* Γ, and an *interface* I. An *attribute environment* $\Gamma : \mathscr{A} \nrightarrow \mathscr{V}$ is a partial map from attribute $a \in \mathscr{A}$ to values $v \in \mathscr{V}$ where $\mathscr{A} \cap \mathscr{V} = \emptyset$. A value could be a number, a name (string), a tuple, etc., An *interface* $I \subseteq \mathscr{A}$ consists of a set of *attributes* that are exposed by a component to control the interactions with other components. We will refer to the attributes in I as *public attributes*, and to those in $dom(\Gamma) - I$ as *private attributes*. During interaction, a component exposes the environment $\Gamma \downarrow I$ which represents the portion of the Γ that can be perceived by the context. It can be obtained from the local Γ by limiting its domain to the attributes in the interface I as defined below:

$$(\Gamma \downarrow I)(a) = \begin{cases} \Gamma(a) & a \in I \\ \bot & \text{otherwise} \end{cases}$$

Components are composed with parallel operator $C_1 \| C_2$.

$$C ::= \Gamma :_I P \quad | \quad C_1 \| C_2$$

Example (step 1/7): Each vertex is modelled in *AbC* as a component of the form $C_i = \Gamma_i :_{\{\mathsf{id},\mathsf{N}\}} P_C$. Public attributes id and N are used to represent the vertex *id* and the set of *neighbours* N, respectively. The overall system is defined as the parallel composition of existing components (i.e. $C_1 \| C_2 \| , \ldots , \| C_n$).

The attribute environment of a vertex Γ_i relies on the following attributes to control the behaviour of a vertex: The attribute "round" stores the current round, while "used" is a set registering the colours used by neighbours. The attribute "counter" counts the number of messages collected by a component, while "send" is used to enable/disable forwarding of messages to neighbours. Attribute "assigned" indicates if a vertex is assigned a colour while "colour" is a colour proposal. These attributes initially have the following values:
round $= 0$, used $= \emptyset$, send $=$ tt, and assigned $=$ ff.

Note that no global knowledge is required and new values for these attributes can only be learnt by means of message exchange among vertices. Also the fact that a vertex knows its neighbours is example-dependent. Thanks to predicates, fully anonymous interactions can be modelled in *AbC*. □

The behaviour of an *AbC* process can be generated by the following grammar:

$$P ::= 0 \mid \alpha.U \mid \langle \Pi \rangle P \mid P_1 + P_2 \mid P_1 | P_2 \mid K$$
$$U ::= [a := E]U \mid P$$

The process 0 denotes the inactive process; $\alpha.U$ denotes a process that executes action α and (possibly) the attribute updates in U and continues as P. Note that the attribute updates $[a := E]U$ are applied instantaneously with the action preceding them. The attribute environment is thus updated by setting the value of attribute a to the evaluation of the expression E. The term $\langle \Pi \rangle P$ is an awareness process, it blocks the execution of process P until the predicate Π evaluates to true; the processes $P_1 + P_2$, $P_1 | P_2$, and K are standard for nondeterminism, parallel composition, and process definition, respectively. The parallel operator "|" does not allow communication between P_1 and P_2 they can only interleave, while the parallel operator "‖" at the component level allows communication between components. The expression this.b denotes the value of attribute b in the current component.

Example (step 2/7): Process P_C specifying the behaviour of a vertex is now defined as the parallel composition of these four processes: $P_C \triangleq F \mid T \mid D \mid A$.

Process F forwards *try*-messages to neighbours, T handles *try*-messages, D handles *done*-messages, and A is used for assigning a final colour. □

The *AbC* communication actions ranged by α can be either $(\tilde{E})@\Pi$ or $\Pi(\tilde{x})$. The construct $(\tilde{E})@\Pi$ denotes an output action, it evaluates the sequence of expressions $\tilde{E}$ under the local attribute environment and then sends the result to the components whose attributes satisfy the predicate Π. Furthermore, $\Pi(\tilde{x})$ denotes an input action, it binds to sequence $\tilde{x}$ the corresponding received values from components whose *communicated attributes* or values satisfy Π.

Example (step 3/7): We further specify process F and a part of process T.

$$F \triangleq \langle \mathsf{send} \wedge \neg \mathsf{assigned} \rangle \\ (\text{“}try\text{”}, \min\{i \notin \texttt{this}.used\}, \texttt{this.round})@ \\ (\texttt{this}.id \in \mathsf{N}).[\mathsf{send} := \mathsf{ff}, \\ \mathsf{colour} := \min\{i \notin \texttt{this.used}\}]F$$

$$T \triangleq ((x\text{=“}try\text{”}) \wedge (\texttt{this.id} > \mathsf{id}) \wedge (\texttt{this.round} = z)) \\ (x, y, z).[\mathsf{counter} := \mathsf{counter} + 1]T + \ldots$$

In process F, when the value of attribute send becomes true, a new colour is selected, send is turned off, and a message containing this colour and the current round is sent to all the vertices having this.id as neighbour. The new colour is the smallest colour that has not yet been selected by neighbours, that is $\min\{i \notin \texttt{this.used}\}$. Remember that attribute used is local and initially is empty and thus all colours are available for a vertex initially. Furthermore, a vertex may only update the value of used at run-time when it receives messages informing that a specific colour is selected. The guard ¬assigned is used to make sure that vertices with assigned colours do not take part in the colour selection anymore.

Process T receives messages of the form (“try", c, r). If $r =$ this.round, then the received message was originated by a vertex performing the same round of the algorithm. The condition this.id $>$ id means that the sender has an *id* smaller than the *id* of the receiver. In this case, the message is ignored (there is no conflict), simply the counter of collected messages (this.counter) is incremented. Other cases, not reported here, e.g. this.id $<$ id, the received colour is recorded to check the presence of conflicts. Note that since the id of the sender is an interface attribute (i.e. id $\in I$), it is automatically added to the message being communicated and this is the reason why the receiver can predicate on the identity of the sender. □

***AbC* Semantics.** The main semantics rules of *AbC* are reported in Table 1. Rule Comp states that a component evolves with (send $\Gamma \triangleright \overline{\Pi}(\tilde{v})$ or receive $\Gamma \triangleright \Pi(\tilde{v})$, denoted by λ) if its internal behaviour, denoted by the relation $\mapsto$, allows it. Rule fComp states that a component can discard a message $\Gamma \triangleright \Pi(\tilde{v})$ if its internal behaviour does not allow

Table 1 *AbC* communication rules

$$\frac{\Gamma :_I P \xmapsto{\lambda} \Gamma' :_I P'}{\Gamma :_I P \xrightarrow{\lambda} \Gamma' :_I P'}\ \textsc{Comp} \qquad \frac{\Gamma :_I P \xmapsto{\Gamma' \triangleright \widetilde{\Pi'}(\tilde{v})} \Gamma :_I P}{\Gamma :_I P \xrightarrow{\Gamma' \triangleright \Pi'(\tilde{v})} \Gamma :_I P}\ \textsc{FComp}$$

$$\frac{C_1 \xrightarrow{\Gamma \triangleright \Pi(\tilde{v})} C_1' \quad C_2 \xrightarrow{\Gamma \triangleright \Pi(\tilde{v})} C_2'}{C_1 \parallel C_2 \xrightarrow{\Gamma \triangleright \Pi(\tilde{v})} C_1' \parallel C_2'}\ \textsc{Sync}$$

$$\frac{C_1 \xrightarrow{\Gamma \triangleright \overline{\Pi}(\tilde{v})} C_1' \quad C_2 \xrightarrow{\Gamma \triangleright \Pi(\tilde{v})} C_2'}{C_1 \parallel C_2 \xrightarrow{\Gamma \triangleright \overline{\Pi}(\tilde{v})} C_1' \parallel C_2'}\ \textsc{ComL} \qquad \frac{C_1 \xrightarrow{\Gamma \triangleright \Pi(\tilde{v})} C_1' \quad C_2 \xrightarrow{\Gamma \triangleright \overline{\Pi}(\tilde{v})} C_2'}{C_1 \parallel C_2 \xrightarrow{\Gamma \triangleright \overline{\Pi}(\tilde{v})} C_1' \parallel C_2'}\ \textsc{ComR}$$

the reception of this message by generating the discarding label $\Gamma \triangleright \widetilde{\Pi}(\tilde{v})$. Rule ComL states that if C_1 evolves to C_1' by sending a message $\Gamma \triangleright \overline{\Pi}(\tilde{v})$, then this message should be delivered to C_2 which evolves to C_2' as a result. Rule ComR is the symmetric rule of ComL. Note that C_2 can be also a parallel composition of different components. Thus, rule Sync states that multiple components can be delivered the same message in a single transition.

The semantics of the parallel composition operator, in rules ComL, ComR, and Sync in Table 1, abstracts from the underlying coordination infrastructure that mediates the interactions between components and thus the semantics assumes atomic message exchange. This implies that no component can evolve before the sent message is delivered to all components executing in parallel. Individual components are in charge of using or discarding incoming messages. Message transmission is non-blocking, but reception is not. For instance, a component can still send a message even if there is no receiver (i.e. all the target components discard the message); a receive operation can, instead, only take place through synchronisation with an available message. However, if we want to use the attribute-based paradigm to program the interactions of distributed applications, atomicity and synchrony are neither efficient nor applicable.

The focus of this article is on providing an *efficient distributed* coordination infrastructure that behaves in agreement with the parallel composition operator of *AbC*. Thus in Table 1, we only formalise the external behaviour of a component, i.e. its ability to send and receive. We show in the example below how interactions are derived based on internal behaviour.

Example (step 4/7): Consider the vertices C_1, C_2, and C_3 where $\Gamma_2(\mathsf{N}) = \{3\}$, $\Gamma_3(\mathsf{N}) = \{1, 4\}$, and $\Gamma_3(\mathsf{id}) = 3$. Now C_1 sent a try message:

$$\Gamma_1 :_{\{1,\{3\}\}} P_C \xrightarrow{\{(\mathsf{id},1),(\mathsf{N},\{3\})\} \triangleright \overline{(1 \in \mathsf{N})}(\text{“}try\text{”},3,5)} \overbrace{\Gamma_1[\mathsf{colour} \leftarrow 3, \mathsf{send} \leftarrow \mathsf{ff}] :_{\{1,\{3\}\}} P_C'}^{C_1'}$$

We have that C_2 discards this message because it is not a neighbour $\Gamma_2 \downarrow I \not\models (1 \in \mathsf{N})$, i.e. $1 \notin \Gamma_2(\mathsf{N})$ because $\Gamma_2(\mathsf{N}) = \{3\}$. Furthermore C_3 accepts the message because $1 \in \Gamma_3(\mathsf{N})$. The system evolves with rule ComL as follows:

$$C_1 \| C_2 \| C_3 \xrightarrow{\{(\mathsf{id},1),(\mathsf{N},\{3\})\} \triangleright \overline{(1 \in \mathsf{N})}(\text{“}try\text{”},3,5)} C_1' \| C_2 \| C_3'$$

□

3 A distributed coordination infrastructure

In this section, we consider a tree-based coordination infrastructure that we have also implemented in Google Go. This infrastructure is introduced to behave in agreement with the parallel composition operator of *AbC* while allowing components to execute asynchronously. Our approach consists of labelling each message with an id that is uniquely identified at the infrastructure level. Components execute asynchronously, while the semantics of the parallel composition operator is preserved by relying on the unique identities of exchanged messages. In essence, if a component wants to send a message, it sends a request to the infrastructure for a fresh id. The infrastructure replies back with a fresh id, and then the component sends a data (the actual) message with the received id. A component receives a data message only when the difference between the incoming data message id and the id of the last received data message equals 1. Otherwise, the data message is added to the component waiting queue until the condition is satisfied.

In this article, we give a full formal account of the tree infrastructure and also investigate its correctness with respect to the semantics of the parallel composition operator of *AbC*. Further details regarding other alternatives can be found in [7]. For the sake of completeness, we will briefly describe these infrastructures. The reason of our focus on the tree infrastructure is because that the tree is more efficient and theoretically more challenging.

3.1 Cluster-based infrastructure

An example of a cluster infrastructure is reported in Fig. 1a. It is composed of a set of server nodes $\mathcal{S}$, sharing a counter ctr for sequencing messages and one input FIFO queue $\mathcal{I}$ to store messages sent by any component C. Cluster nodes can have exclusive locks on both the cluster's counter and the queue. Components register directly to the cluster (i.e. their addresses are registered in $\mathcal{D}$) and send messages to be added to the input FIFO queue. When a server node retrieves a request from the cluster queue, it replies to the requester with the value of the cluster counter. By doing so, the cluster counter is incremented. If a server retrieves a data message, it forwards the message to all components in the cluster except for the sender.

3.2 Ring-based infrastructure

An example of a ring infrastructure is reported in Fig. 1c. It is composed of a set of server nodes $\mathcal{S}$, organised in a logical ring and sharing a counter ctr for sequencing messages coming from components. Each node manages a group of components (i.e. unlike cluster nodes, each ring node has a dedicated registration queue $\mathcal{D}$) and can have exclusive locks to the ring counter. When a request message arrives to a node from one of its components, the node acquires a lock on the ring counter, copies it current value, releases it after incrementing it by 1, and finally sends a reply, carrying a fresh id, to the requester. Data messages are directly added to the node's waiting queue and will be only forwarded to the node's components and to the neighbour node when all previous messages (i.e. with a smaller id) are received.

3.3 A tree-based infrastructure

An example of a tree infrastructure is reported in Fig 1c. It is composed of a set of servers $\mathcal{S}$, organised in a logical tree. A component C can be connected to one server (its parent) in the tree and can interact with others in any part of the tree by only dealing with its parent. When a component wants to send a message, it asks for a fresh id from its parent. If the parent is the root of the tree, it replies with a fresh id; otherwise, it forwards the message to its own parent in the tree. Only the root of the tree can sequence messages. As the case with the ring, each tree node has a registration queue $\mathcal{D}$ and is responsible for only a group of components.

We would like to mention that our results in this article do not consider reliability of the communication infrastructures. We only focus on the efficiency and the correctness of coordination, and we leave the reliability issue open for future work. We strongly believe that standard approaches that consider replication of server nodes can be used to guarantee the overall reliability of the considered infrastructures. However, an interesting direction would be to consider infrastructures that adapt to failure and reconfigure themselves to maintain their reliability, see [8,11].

In the following, we provide a full formal account of the tree infrastructure.

3.4 Preliminary notations and definitions

We report the required set of notations and definitions we use to formalise the semantics of the tree infrastructure. Furthermore we define a general definition of an infrastructure component. The idea is to keep the infrastructure component totally decoupled from the type of the infrastructure it is connected to. This way the behaviour of a component will be totally independent from the one of the infrastructure.

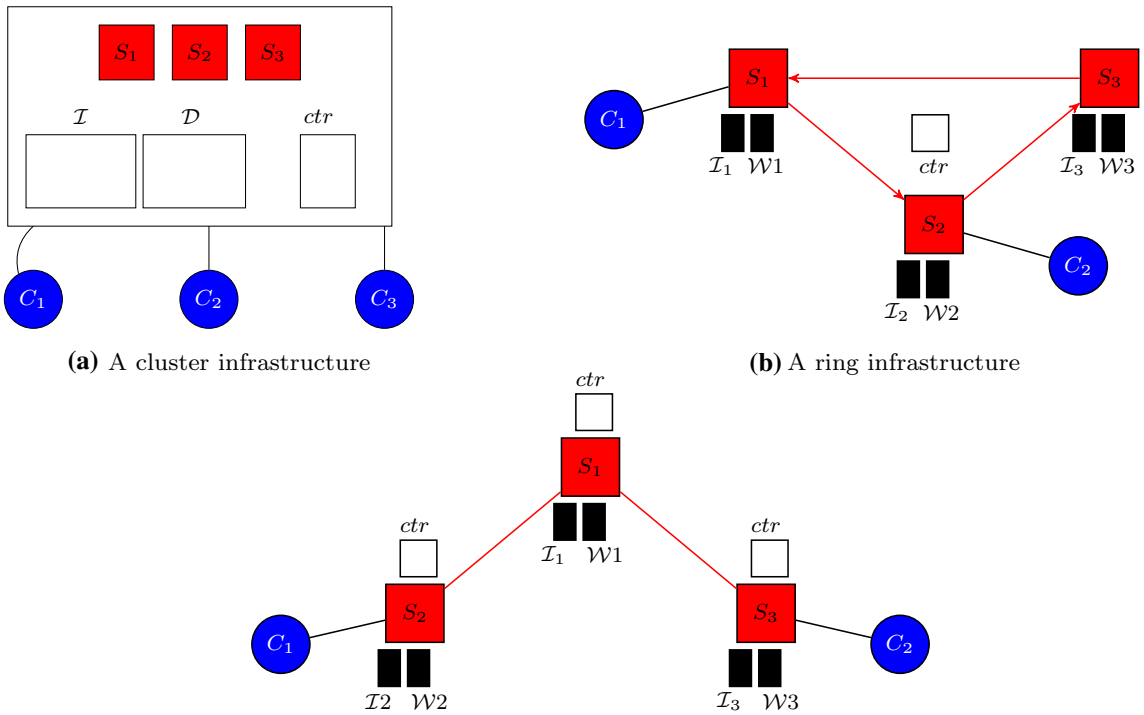
We use the following definition of a *Configuration* to provide a compact semantics. For clarity, we postfix the configuration of a component, an infrastructure, and a server with the letters a, n, and s, respectively.

Definition 1 (*Configuration*) A configuration C, is a set of the form $C = \langle c_1, \ldots, c_n \rangle$. The symbol '…' is formally regarded as a meta-variable ranging over unmentioned elements of the configuration. The explicit '…' is obligatory, and ensures that unmentioned elements of a configuration are never excluded, but they do not play any role in the current context. Different occurrences of '…' in the same context stand for the same set of unmentioned elements. This definition is borrowed from Peter Mosses' style of defining labels in modular structure operational semantics [29].

We use the reduction relation $\rightsquigarrow_{\triangleright} \subseteq \textsc{Cfig} \times \textsc{Lab} \times \textsc{Cfig}$ to define the semantics of a configuration where Cfig denotes the set of configurations, Lab denotes the set of reduction labels which can be a message m, a silent transition τ, or an empty label, and $\rightsquigarrow_{\triangleright}^{*}$ denotes the transitive closure of $\rightsquigarrow_{\triangleright}$. Moreover, we will use the following notations:

- We have two kinds of messages, an *AbC* message '$_{\text{msg}}$' (i.e. $\Gamma \triangleright \Pi(\tilde{v})$) and an infrastructure message 'm'; the latter can be of three different templates: (i) request {'Q', route, dest}, (ii) reply {'R', id, route, dest}, and (iii) data {'D', id, src, dest, msg}. The route field in a request or a reply message is a linked list containing the addresses of the nodes that the message traversed.
- The notation $\stackrel{?}{=}$ denotes a template matching.
- The notation $T[f]$ denotes the value of the element f in T.
- The notation $m :: \mathcal{W}$ denotes a queue with message m on top of it.

We also use these operations: $L.get()$ returns the element at the front of a list/queue, while $L \leftarrow\!\!| \; m$ returns the



(a) A cluster infrastructure

(b) A ring infrastructure

(c) A tree infrastructure

Fig. 1 Communication infrastructures

list/queue resulting from adding m to the back of L, and $L \backslash x$ removes x from L and returns the rest.

3.5 Infrastructure component

We formally define a general infrastructure component and its external behaviour. In the following sections, we proceed by formally defining the tree infrastructure and its behaviour.

Definition 2 (*Infrastructure component*) An infrastructure component, a, is defined by the configuration: $a = \langle addr, nid, mid, on, \mathcal{W}, \mathcal{X}, G \rangle$ where $addr$ refers to its address, nid (initially 0) refers to the id of the next data message to be received, mid (initially -1) refers to the id of the most recent reply, on (initially 0) indicates whether a request message can be sent. $\mathcal{W}$ is a priority waiting queue where the top of $\mathcal{W}$ is the data message with the least id, and $\mathcal{X}$ refers to the address of the parent server. Furthermore, G ranges over $\Gamma :_I P$ and $[\Gamma :_I P]$ where $[\Gamma :_I P]$ indicates an AbC component in an intermediate state.

The intermediate state, in Definition 2, is important to allow co-located processes (running in the same component) to interleave their behaviours without compromising the semantics, i.e. $[\Gamma :_I P_1 | P_2]$ where P_1 is waiting an id to send and P_2 is willing to receive. Here P_2 can only receive a message if it was able to do so before the intermediate state as we will explain later.

The semantics of an infrastructure component is reported in Table 2. Rule Out states that if the AbC component $\Gamma :_I P$ encapsulated inside an infrastructure component is able to send a message $\Gamma :_I P \xrightarrow{\Gamma' \triangleright \overline{\Pi}(\tilde{v})} \Gamma'' :_I P'$, the flag on is set to 1 and $\Gamma :_I P$ goes into an intermediate state $[\Gamma :_I P]$. Rule Med states that an intermediate state component can only receive a message $\Gamma \triangleright \Pi(\tilde{v})$ if it was able to receive it before the intermediate state. Rule Req states that a component sends a request, to the parent server, only if $on == 1$. In this case, it adds its address to the $route$ of the message and resets on to 0. Rule RcvR states that a component receives a reply if the destination field of the reply matches its address; after that mid gets the value of the id received in the reply. Rule Snd states that a component $\Gamma :_I P$ can send a message $\Gamma' \triangleright \overline{\Pi}(\tilde{v})$ and evolves to $\Gamma'' :_I P'$ only if $nid == mid$; this implies that a fresh id is received ($mid \neq -1$) and all messages with $m[id] < mid$ have been already received. By doing so, an infrastructure data message, with msg field equal to $\Gamma \triangleright \Pi(\tilde{v})$, is sent, nid is incremented, and mid is reset. Rule RcvD states that a component receives a data message from the infrastructure if $m[id] \geq nid$; this is important to avoid duplicate messages. The message is then added to the priority queue, $\mathcal{W}$. Finally, rule Hnd states that when the id of the message on top of $\mathcal{W}$ matches nid, component G is allowed to receive

Table 2 The semantics of a component

$$\frac{\Gamma :_I P \xrightarrow{\Gamma' \triangleright \overline{\Pi}(\tilde{v})} \Gamma'' :_I P'}{\langle on,\ \Gamma :_I P, \ldots \rangle_a \overset{\tau}{\rightsquigarrow\!\triangleright} \langle 1,\ [\Gamma :_I P], \ldots \rangle_a}\ \textsc{Out} \qquad \frac{\Gamma :_I P \xrightarrow{\Gamma' \triangleright \Pi(\tilde{v})} \Gamma'' :_I P'}{[\Gamma :_I P] \xrightarrow{\Gamma' \triangleright \Pi(\tilde{v})} [\Gamma'' :_I P']}\ \textsc{Med}$$

$$\frac{on == 1}{\langle addr, on, \mathcal{X}, \ldots \rangle_a \overset{\{\text{`}Q\text{'},\ \{addr\},\ \mathcal{X}\}}{\rightsquigarrow\!\triangleright} \langle addr, 0, \mathcal{X}, \ldots \rangle_a}\ \textsc{Req}$$

$$\langle addr, mid, \ldots \rangle_a \overset{\{\text{`}R\text{'},\ id,\ \{\},\ addr\}}{\rightsquigarrow\!\triangleright} \langle addr, id, \ldots \rangle_a \quad \textsc{RcvR}$$

$$\frac{nid == mid \qquad \Gamma :_I P \xrightarrow{\Gamma' \triangleright \overline{\Pi}(\tilde{v})} \Gamma'' :_I P'}{\langle addr, nid, mid, [\Gamma :_I P], \mathcal{X}, \ldots \rangle_a \overset{\{\text{`}D\text{'}, mid, addr, \mathcal{X}, \Gamma' \triangleright \Pi(\tilde{v})\}}{\rightsquigarrow\!\triangleright} \langle addr, nid + 1, -1, \Gamma'' :_I P', \mathcal{X}, \ldots \rangle_a}\ \textsc{Snd}$$

$$\frac{m \overset{?}{=} \{\text{`}D\text{'},\ id,\ \mathcal{X},\ addr,\ msg\} \qquad id \geq nid}{\langle addr, nid, \mathcal{W}, \mathcal{X}, \ldots \rangle_a \overset{m}{\rightsquigarrow\!\triangleright} \langle addr, nid, \mathcal{W} \leftarrow m, \mathcal{X}, \ldots \rangle_a}\ \textsc{RcvD}$$

$$\frac{m[id] == nid \qquad G \xrightarrow{m[msg]} G'}{\langle nid, G,\ m :: \mathcal{W}', \ldots \rangle_a \overset{\tau}{\rightsquigarrow\!\triangleright} \langle nid + 1, G',\ \mathcal{W}', \ldots \rangle_a}\ \textsc{Hnd}$$

that message; as a result, nid is incremented by 1 and thus m is removed.

Example (step 5/7): As the semantics suggests, an infrastructure component is an encapsulation of an *AbC* component and thus these details are hidden from the programmer. In other words, the programmer can create an infrastructure component (Definition 2) for our example in Sect. 2 by only specifying the address $\mathcal{X}$ of the parent server, substituting a vertex component $\Gamma_i :_{\{\text{id,N}\}} P_C$ in place of G and set other parts to their initial values as follows:

$$a = \langle addr, nid = 0, mid = -1, on = 0, \mathcal{W} = \{\}, \mathcal{X} = \text{“some address"}, \Gamma_i :_{\{\text{id,N}\}} P_C \rangle$$

Note that in a real application a DHCP server automatically assigns a unique address $addr$ to a component when executed. We will discuss these implementation details in Sect. 4.

3.6 A formal definition of the tree infrastructure

In this section, we formally define the structure of the tree and its semantics. Furthermore we provide a proof of correctness of its behaviour with respect to the semantics of the parallel composition operator of *AbC*.

Definition 3 (*Tree server*) A tree server s is a configuration: $s = \langle addr,\ ctr,\ nid,\ \mathcal{D},\ \mathcal{M},\ \mathcal{I},\ \mathcal{W},\ \mathcal{X} \rangle$ where ctr is a counter to generate fresh ids, $\mathcal{D}$ is a set containing the addresses of the server's children which include connected components and servers, $\mathcal{M}$ is a multicast set (initially $\mathcal{M} = \mathcal{D}$), and $\mathcal{I}$ is a FIFO input queue. The rest are defined as before.

Definition 4 (*Tree infrastructure*) A tree infrastructure $\mathcal{N}$ is defined by the configuration: $\mathcal{N} = \langle \mathcal{S}, \mathcal{A} \rangle$ where $\mathcal{S}$ denotes the set of servers and $\mathcal{A}$ denotes the set of connected components such that:

- $\forall s_1, s_2 \in \mathcal{S}$, we say that s_1 is a direct child of s_2, written $s_1 \prec s_2$, if and only if $s_1[\mathcal{X}] = s_2[addr]$; $\prec^+$ denotes the transitive closure of $\prec$.
- $\forall s \in \mathcal{S}$, we have that $s \not\prec^+ s$.
- The root: $\exists s \in \mathcal{S}$ such that for any $s' \in (\mathcal{S} \backslash \{s\})$, $s' \prec^+ s$ and we have that:
  - $s'[nid] \leq s[ctr]$.
  - For any message $m \in s'[\mathcal{W}]$ we have that $m[id] \leq s[ctr]$.
- A root is unique: if $s, s' \in \mathcal{S}$ and $s[\mathcal{X}] = s'[\mathcal{X}] = \perp$ then we have that $s = s'$.
- $\forall s \in \mathcal{S}$ and for each message $m \in s[\mathcal{W}]$, we have that $m[id] \geq s[nid]$.

The semantics rules of a tree infrastructure are reported in Table 3. The rules (S ↔ S) and (S ↔ A) state that a tree evolves when a message m is exchanged either between two of its servers (s_1 and s_2) or between a server and a component, respectively. Furthermore, the rules (S) and (A) state that a tree evolves when one of its servers or one of its connected components evolves independently.

Table 3 Tree infrastructure semantics

$$\frac{s_1 \overset{m}{\rightsquigarrow\!\triangleright} s_1' \qquad s_2 \overset{m}{\rightsquigarrow\!\triangleright} s_2'}{\langle\{s_1, s_2\} \cup \mathcal{S}', \mathcal{A}\rangle_n \rightsquigarrow\!\triangleright \langle\{s_1', s_2'\} \cup \mathcal{S}', \mathcal{A}\rangle_n}\ \text{S}\leftrightarrow\text{S} \qquad \frac{s \overset{\tau}{\rightsquigarrow\!\triangleright} s'}{\langle\{s\} \cup \mathcal{S}', \mathcal{A}\rangle_n \rightsquigarrow\!\triangleright \langle\{s'\} \cup \mathcal{S}', \mathcal{A}\rangle_n}\ \text{S}$$

$$\frac{s \overset{m}{\rightsquigarrow\!\triangleright} s' \qquad a \overset{m}{\rightsquigarrow\!\triangleright} a'}{\langle\{s\} \cup \mathcal{S}', \{a\} \cup \mathcal{A}'\rangle_n \rightsquigarrow\!\triangleright \langle\{s'\} \cup \mathcal{S}', \{a'\} \cup \mathcal{A}'\rangle_n}\ \text{S}\leftrightarrow\text{A} \qquad \frac{a \overset{\tau}{\rightsquigarrow\!\triangleright} a'}{\langle\mathcal{S}, \{a\} \cup \mathcal{A}'\rangle_n \rightsquigarrow\!\triangleright \langle\mathcal{S}, \{a'\} \cup \mathcal{A}'\rangle_n}\ \text{A}$$

Table 4 Tree server semantics

$$\frac{m[dest] == addr}{\langle addr, \mathcal{I}, \ldots\rangle_s \overset{m}{\rightsquigarrow\!\triangleright} \langle addr, \mathcal{I} \leftarrow\!\shortmid m, \ldots\rangle_s}\ \textsc{In}$$

$$\frac{m \overset{?}{=} \{\text{`}Q\text{'}, route, addr\} \qquad \mathcal{X} == \bot \qquad x = route.get()}{\langle addr, ctr, \mathcal{X},\ m :: \mathcal{I}',\ \ldots\rangle_s \overset{\{\text{`}R\text{'}, ctr, route \setminus x, x\}}{\rightsquigarrow\!\triangleright} \langle addr, ctr + 1, \mathcal{X}, \mathcal{I}', \ldots\rangle_s}\ \textsc{Reply}$$

$$\frac{m \overset{?}{=} \{\text{`}Q\text{'}, route, addr\} \qquad \mathcal{X} \neq \bot \qquad route' = route.add(addr)}{\langle addr, \mathcal{X},\ m :: \mathcal{I}',\ \ldots\rangle_s \overset{\{\text{`}Q\text{'}, route', \mathcal{X}\}}{\rightsquigarrow\!\triangleright} \langle addr, \mathcal{X}, \mathcal{I}', \ldots\rangle_s}\ \textsc{qFwd}$$

$$\frac{m \overset{?}{=} \{\text{`}R\text{'}, id, route, addr\} \qquad x = route.get()}{\langle addr,\ m :: \mathcal{I}',\ \ldots\rangle_s \overset{\{\text{`}R\text{'}, id, route \setminus x, x\}}{\rightsquigarrow\!\triangleright} \langle addr, \mathcal{I}', \ldots\rangle_s}\ \textsc{rFwd}$$

$$\frac{m \overset{?}{=} \{\text{`}D\text{'}, id, addr', addr, msg\} \qquad (\mathcal{X} == addr' \vee \mathcal{X} == \bot)}{\langle addr, \mathcal{X}, \mathcal{W},\ m :: \mathcal{I}', \ldots\rangle_s \overset{\tau}{\rightsquigarrow\!\triangleright} \langle addr, \mathcal{X}, \mathcal{W} \leftarrow\!\shortmid m,\ \mathcal{I}', \ldots\rangle_s}\ \textsc{wIn}$$

$$\frac{m \overset{?}{=} \{\text{`}D\text{'}, id, addr', addr, msg\} \qquad (\mathcal{X} \neq addr' \wedge \mathcal{X} \neq \bot)}{\langle addr, \mathcal{X}, \mathcal{W},\ m :: \mathcal{I}', \ldots\rangle_s \overset{\{\text{`}D\text{'}, id, addr, \mathcal{X}, msg\}}{\rightsquigarrow\!\triangleright} \langle addr, \mathcal{X}, \mathcal{W} \leftarrow\!\shortmid m, \mathcal{I}', \ldots\rangle_s}\ \textsc{wNxt}$$

$$\frac{|\mathcal{M}| > 1 \qquad \begin{array}{c} m[id] == nid \\ m \overset{?}{=} \{\text{`}D\text{'}, id, addr', addr, msg\} \end{array} \qquad \mathcal{M}' = \mathcal{M} \setminus addr' \qquad x = \mathcal{M}'.get()}{\langle addr, nid, \mathcal{D}, \mathcal{M},\ m :: \mathcal{W}', \ldots\rangle_s \overset{\{\text{`}D\text{'}, id, addr, x, msg\}}{\rightsquigarrow\!\triangleright} \langle addr, nid, \mathcal{D}, \mathcal{M}' \setminus x,\ m :: \mathcal{W}', \ldots\rangle_s}\ \textsc{dFwd}$$

$$\frac{|\mathcal{M}| = 1 \qquad \mathcal{M}.get() \neq addr' \qquad m[id] == nid \qquad m \overset{?}{=} \{\text{`}D\text{'}, id, addr', addr, msg\} \qquad x = \mathcal{M}.get()}{\langle addr, nid, \mathcal{D}, \mathcal{M},\ m :: \mathcal{W}', \ldots\rangle_s \overset{\{\text{`}D\text{'}, id, addr, x, msg\}}{\rightsquigarrow\!\triangleright} \langle addr,\ nid + 1, \mathcal{D}, \mathcal{D}, \mathcal{W}', \ldots\rangle_s}\ \textsc{eFwd}$$

$$\frac{|\mathcal{M}| = 1 \qquad \mathcal{M}.get() = addr' \qquad m[id] == nid \qquad m \overset{?}{=} \{\text{`}D\text{'}, id, addr', addr, msg\}}{\langle addr, nid, \mathcal{D}, \mathcal{M},\ m :: \mathcal{W}', \ldots\rangle_s \overset{\tau}{\rightsquigarrow\!\triangleright} \langle addr,\ nid + 1, \mathcal{D}, \mathcal{D}, \mathcal{W}', \ldots\rangle_s}\ \textsc{nFwd}$$

The semantics rules of a tree server are defined by the rules in Table 4. Rule In states that a server receives a message m and adds it to the back of its input queue ($\mathcal{I} \leftarrow\!\shortmid m$) if the destination field of m matches its own address $addr$. Rule Reply states that if a root server gets a request from the front of its input queue $m :: \mathcal{I}'$, it sends a reply to the requester by getting its address from the route of the message $x = route.get()$. The id of the reply is assigned the value of the root's counter ctr. By doing so, the counter is incremented. On the other hand, a non-root server adds its address to the message's route and forwards it to its parent as stated by rule qFwd. Rule rFwd instead is used for forwarding reply messages. Rule wIn states that if a server gets a data message from its input queue $\mathcal{I}$ and it is the root or its parent is the source of the message (i.e. $\mathcal{X} == addr' \vee \mathcal{X} == \bot$), the server evolves silently and the message is added to its waiting queue. If the condition ($\mathcal{X} == addr' \vee \mathcal{X} == \bot$) does not hold, the message is also forwarded to the parent as stated by

rule WNXT. Furthermore, rule DFWD states that when the id of the message on top of $\mathcal{W}$ matches nid (i.e. $m[id] == nid$), the server starts forwarding m to its children one by one except for the sender. Note that this rule can be applied many times as long as the multicast set $\mathcal{M}$ contains more than one element, i.e. $|\mathcal{M}| > 1$. Once $\mathcal{M}$ has only one element, rule EFWD is applied to forward the message to the last address in $\mathcal{M}$. As a result, nid is incremented, m is removed from $\mathcal{W}$, and the multicast set $\mathcal{M}$ is reset to its initial value. Note that rule NFWD handles the case when $\mathcal{M}$ has only the address of the sender. Thus the message is discarded as the sender cannot receive its own message.

Correctness. Since there is a single sequencer in the tree, i.e. the root, two messages can never have the same id. We only need the following results to ensure that the tree behaves in agreement with the *AbC* parallel composition operator. In essence, Proposition 2 ensures that if any component in the tree sends a request for a fresh id, it will get it. Proposition 4 ensures that any two components in the tree with different nid will converge to the same one. However, to prove Proposition 4, we need to prove Lemma 1 and Proposition 3 which guarantee the same results among tree' servers. This implies that messages are delivered to all components. Proposition 5 instead ensures that no message stays in the waiting queue indefinitely.

Proposition 1 *For each pair of nodes $s_1, s_2 \in \mathcal{N}[\mathcal{S}]$, if s_1 sends a request to s_2, then eventually s_2 will send a reply to s_1.*

Proof The proof proceeds by induction on the level of s_2 in the tree, $L(s_2)$.

- Base case: $L(s_2) = 0$, this implies that node s_2 is the root of the tree (i.e. $s_1[\mathcal{X}] = s_2[addr]$). By Table 4, rule QFWD, when s_1 gets a request from its input queue, it adds its address to the route of the message and forwards it to its next node, in this case and by Table 4, rule IN, s_2 adds the request to its input queue. The overall infrastructure evolves by applying rule $s \leftrightarrow s$, Table 3. The tree $\mathcal{N}$ evolves by multiple applications of the rules in Table 3 until s_2 gets the request from its input queue. It labels it with a fresh id and sends a reply back to the requester by applying rule REPLY and s_1 receives the reply by applying rule IN, Table 4. So we have that $\mathcal{N} \rightsquigarrow\triangleright^* \mathcal{N}'$ we have that s_2 will eventually send a reply back to s_1.
- Inductive hypothesis: $L(s_2) \leq k$, if s_1 sends a request to s_2, then s_2 will eventually reply to s_1.
- Inductive step: Now it is sufficient to prove the claim for s_2, where $L(s_2) = k + 1$. By Table 4, rule QFWD, when s_1 gets a request from its input queue, it adds its address to the route of the message and forwards it to s_2. We have that according to Table 3 $\mathcal{N} \rightsquigarrow\triangleright^* \mathcal{N}'$ and s_2 can also forward the request to its next node, say s' where $L(s') = k$, by Table 4, rule QFWD. By the induction hypothesis, we have that s' will eventually send a reply to s_2 since the claim holds for level k. When the reply arrives to s_2, it will send it to s_1 by applying rule RFWD as required.

□

Proposition 2 *For any component, with address $addr$ and a parent $\mathcal{X}$, connected to a tree infrastructure $\mathcal{N}$, we have that if*

$$\langle addr, on, \mathcal{X}, \ldots\rangle_a \overset{\{\text{`Q'},\{addr\},\mathcal{X}\}}{\rightsquigarrow\triangleright} \langle addr, 0, \mathcal{X}, \ldots\rangle_a$$

then $\mathcal{N} \rightsquigarrow\triangleright^* \mathcal{N}'$ and

$$\langle addr, mid, \ldots\rangle_a \overset{\{\text{`R'},id,\{\},addr\}}{\rightsquigarrow\triangleright} \langle addr, id, \ldots\rangle_a$$

Proof A component a_1 can send a request by applying rule REQ, Table 2 and its node, say s_1, can receive it by applying rule IN, Table 4 and adds it to its input queue. The tree evolves with rule $s \leftrightarrow a$, Table 3. By relying on Proposition 1 and $\mathcal{N} \rightsquigarrow\triangleright^* \mathcal{N}'$, s_1 will send a reply back to a_1 by applying either rule REPLY or RFWD, Table 4. On the other hand, a_1 will receive the reply by applying rule RCVR and the tree evolves with rule $s \leftrightarrow a$, Table 3 as required. □

To prove Proposition 3, we need first to prove the following lemma. This lemma ensures that adjacent nodes will eventually converge to the same nid. In other words, a node must forward the removed data message from its queue to all immediate neighbours (i.e. tree nodes in its $\mathcal{D}$) before incrementing its nid.

Lemma 1 *For every two tree nodes s_1 and s_2 and a tree-based infrastructure $\mathcal{N}$ such that $s_1, s_2 \in \mathcal{N}[\mathcal{S}]$, we have that:*

- *If $s_1 \prec s_2 \wedge s_1[nid] < s_2[nid]$ then $\mathcal{N} \rightsquigarrow\triangleright^* \mathcal{N}'$ and $s_1[nid] = s_2[nid]$.*
- *If $s_2 \prec s_1 \wedge s_1[nid] < s_2[nid]$ then $\mathcal{N} \rightsquigarrow\triangleright^* \mathcal{N}'$ and $s_1[nid] = s_2[nid]$.*

Proof The proof proceed by induction on the difference between $s_2[nid]$ and $s_1[nid]$. We only prove the first statement as the second one is analogous.

- Base case, $s_2[nid] - s_1[nid] = 1$: From Table 4, rule EFWD, nid is only incremented after message m, where $m[id] = s_2[nid] - 1$, is forwarded to the last child in $\mathcal{D}$. It can also be incremented when the message is discarded by rule NFWD to avoid sending the message back to the sender. But $s_1 \prec s_2$, so we know that s_1 already received m and added m to its input queue by rule IN. Thus $\mathcal{N} \rightsquigarrow\triangleright$

$^*\mathcal{N}'$ and finally m moves from the input queue to the waiting queue i.e. $s_1[\mathcal{W}] = m :: q$ by either rule WIN or rule WNXT. Note that this is a priority queue that sorts its messages according to their identities. Since $s_2[nid] - s_1[nid] = 1$, we have that $s_2[nid] - 1 = s_1[nid] = m[id]$. This means that m is ordered with respect to s_1. By multiple applications of rule DFWD and followed by one application of rule EFWD and/or NFWD, Table 4 to forward m to children, we have that $s_1[nid] = s_1[nid] + 1 = s_2[nid]$ as required.

- Inductive hypothesis: $s_2[nid] - s_1[nid] \leq k$ given that $s_1 \prec s_2$, then $\mathcal{N} \rightsquigarrow\!\triangleright^*\mathcal{N}'$ and $s_1[nid] = s_2[nid]$.
- Inductive step: Now it is sufficient to prove the claim for s_1 and s_2 such that $s_2[nid] - s_1[nid] = k + 1$.
  From Table 4, rules EFWD and NFWD, we know that message m, where $m[id] = s_2[nid] - 1$, has been already forwarded to children in $\mathcal{D}$ and/or to next node by rule WNXT and s_1 already received m and added m to its input queue. We have that $\mathcal{N} \rightsquigarrow\!\triangleright^*\mathcal{N}'$ and finally m moves from the input queue to the waiting queue by either rule WIN or rule WNXT. Since $s_2[nid] - s_1[nid] = k + 1$ then $m[id] - s_1[nid] = k$. This means that k-messages from s_2 already exist in the queue of s_1 and need to be processed first and then after m can be processed. By the induction hypothesis, $\mathcal{N} \rightsquigarrow\!\triangleright^*\mathcal{N}'$ and $s_1[nid] = s_2[nid]$ where $s_2[nid] - s_1[nid] \leq k$. So we have that $s_1[nid] = s_1[nid] + k$, but $m[id] - s_1[nid] = k$. This implies that $m[id] = s_1[nid]$, i.e. message m is ordered with respect to s_1. Now again by Table 4, multiple applications of rule DFWD followed by one application of rule EFWD and/or NFWD, $\mathcal{N}' \rightsquigarrow\!\triangleright^*\mathcal{N}''$, $s_1[nid] = s_1[nid] + 1 = s_2[nid] = s_1[nid] + k + 1$ and $s_2[nid] = s_1[nid]$ as required. □

Proposition 3 *Let s_1 and s_2 be two tree nodes and $\mathcal{N}$ be a tree-based infrastructure, $\forall s_1, s_2 \in \mathcal{N}[\mathcal{S}] \wedge s_1[nid] \prec s_2[nid]$, we have that $\mathcal{N} \rightsquigarrow\!\triangleright^*\mathcal{N}'$ and $s_1[nid] = s_2[nid]$.*

Proof The proof proceeds by case analysis on $s \prec^+ s'$. Since the topology of the infrastructure is tree based, we have three cases.

- Case 1, $s_1 \prec^+ s_2$: This case can be proved by induction on the distance, $d(s_1, s_2)$, between s_1 and s_2 in the tree. Function $d(s_1, s_2)$ is defined inductively as follows:

$$d(s_1, s_2) = \begin{cases} 0, & \text{for } s_1 \prec s_2 \text{ or } s_2 \prec s_1 \\ 1 + d(t', s_2), & \text{for } s_1 \prec^+ s_2 \text{ where } s_1 \prec t' \\ 1 + d(s_1, t'), & \text{for } s_2 \prec^+ s_1 \text{ where } s_2 \prec t' \end{cases}$$

  - Base case, $d(s_1, s_2) = 0$: directly from Lemma 1.
  - Inductive hypothesis:
    Suppose that $\forall s_1, s_2 \in \mathcal{N}[\mathcal{S}]: d(s_1, s_2) \leq k$ where $k > 0$ and given $s_1[nid] < s_2[nid]$, we have that $\mathcal{N} \rightsquigarrow\!\triangleright^*\mathcal{N}'$ and $s_1[nid] = s_2[nid]$.
  - Inductive step: Now it is sufficient to prove the claim for s_1 and s_2 where $d(s_1, s_2) = k + 1$.
    From Lemma 1, for s_2 at distance $k + 1$ from s_1 and s_3 at distance k from s_1, i.e. $s_1 \prec^+ s_3 \prec s_2$, we have that $\mathcal{N} \rightsquigarrow\!\triangleright^*\mathcal{N}'$ and $s_3[nid] = s_2[nid]$. But $d(s_1, s_3) = k$, so by the induction hypothesis we have that $\mathcal{N}' \rightsquigarrow\!\triangleright^*\mathcal{N}''$ and $s_1[nid] = s_3[nid] = s_2[nid]$ as required.

- Case 2, $s_2 \prec^+ s_1$: is analogous to the previous case.
- Case 3, $\exists s_3 : s_1 \prec^+ s_3 \wedge s_2 \prec^+ s_3$: we have several cases, but here we only consider one case and the others follow in a similar way. If $s_1[nid] > s_3[nid] < s_2[nid]$, we first take s_3 and s_2 and by Case 2, we have that $\mathcal{N} \rightsquigarrow\!\triangleright^*\mathcal{N}'$ and $s_3[nid] = s_2[nid] > s_1[nid]$. Now for s_1 and s_3 and by Case 1, we have that $\mathcal{N}' \rightsquigarrow\!\triangleright^*\mathcal{N}''$ and $s_1[nid] = s_2[nid]$ as required. □

Proposition 4 *Given any two components a_1 and a_2 in a tree infrastructure $\mathcal{N}$ such that $a_1[nid] < a_2[nid]$, we have that $\mathcal{N} \rightsquigarrow\!\triangleright^*\mathcal{N}'$ and $a_1[nid] = a_2[nid]$.*

Proof The proof follows directly by Proposition 3 and the semantics rules in Tables 2 and 3. □

Proposition 5 *Given a tree infrastructure $\mathcal{N} = \langle \mathcal{S}, \mathcal{A} \rangle$, for any $c \in \mathcal{S} \cup \mathcal{A}$ where $c[\mathcal{W}] = m :: \mathcal{W}'$, we have that $\mathcal{N} \rightsquigarrow\!\triangleright^*\mathcal{N}'$ and $c[\mathcal{W}] = \mathcal{W}' +\!\!+ \mathcal{W}''$ where $+\!\!+$ returns a priority queue composed by the sub-queues $\mathcal{W}'$ and $\mathcal{W}''$.*

Proof The proof follows from Propositions 2, 3 and 4, and by induction on the difference between $c[nid]$ and $m[id]$. □

Proposition 5 states that a message m on top of the waiting queue $\mathcal{W}$ of a component cannot stay forever as the infrastructure $\mathcal{N}$ evolves. The component processes m and removes it from $\mathcal{W}$ during the infrastructure evolution. While doing so, other messages (represented by a sub-queue $\mathcal{W}''$) may arrive to the end of $\mathcal{W}$.

4 A go attribute-based interaction API

GoAt[1] is a *distributed* programming API for supporting attribute-based interaction directly in Google Go. Go is a new programming language, developed by Google to handle the complexities of networked systems and massive computation clusters, and to make working in these environments more productive. It supports concurrency and inter-process communication through a compact set of powerful primitives and lightweight processes, called *goroutines*.

[1] https://giulio-garbi.github.io/goat/.

As opposed to Java, Go provides an intuitive and light weight concurrency model with a well-understood semantics. It extends the CSP model [23] with mobility by allowing channel-passing, like in π-calculus [28]. However, channel-passing in Go is possible only locally between goroutines. Go also supports buffered channels with a finite size. When the buffer size is 0, goroutines block execution and can only communicate by means of synchronisation. Otherwise, channels behave like mailboxes in Erlang which is, however, actor-based [5], and for interaction, it relies on identities rather than on channels. Note that such concurrency features of Go can be mimicked with the `core.async` concurrency library of Clojure. However, the main difference is that concurrency in Go is an actual ingredient of the language rather than an integrated library. Also note that Clojure way of dealing with concurrency is similar to the one of Java and thus without the `core.async` library, concurrent programming in Clojure is not an ideal choice. On the other hand, Clojure is less verbose and has a clean typing system when compared to Go. Thus using Clojure with the `core.async` library provides an elegant and clean environment to write concurrent programs. Another candidate would be using the Akka concurrency library in Scala. Akka handles concurrency based on the Actor model as the case of Erlang. However, being integrated in a rich and functional language like Scala, it unlocks the reuse of Scala internal DSL features and easily supports interoperability of Java and other JVM-based languages.

All of these alternatives have their pros and cons, and it would be interesting as a future work to find out what class of applications each one of them handles the best. However, since this is out of the scope of this article, we only focus on the Go language. We believe that the generality, efficiency and the clean concurrency model of Go make it an appropriate language for programming CAS. Thus, we integrated attribute-based interaction in Go via the *distributed GoAt* API to move the mobility of Go concurrency to the next level.

In what follows, we present the actual implementation of the distributed coordination infrastructures in Google Go and present the syntax of the *GoAt* API.

4.1 A Go implementation of infrastructures

In this section, we consider a Go implementation of three distributed coordination infrastructures for managing message exchange of the *GoAt* API. These infrastructures model faithfully the parallel composition operator of the *AbC* calculus.

The projection of a *GoAt* system with respect to a specific component is reported in Fig. 2. It mainly consists of three parts: (i) component, (ii) agent, and (iii) infrastructure. The agent provides a standard interface between a *GoAt* component and the underlying coordination infrastructure and mediates message exchange between them. Actually, the

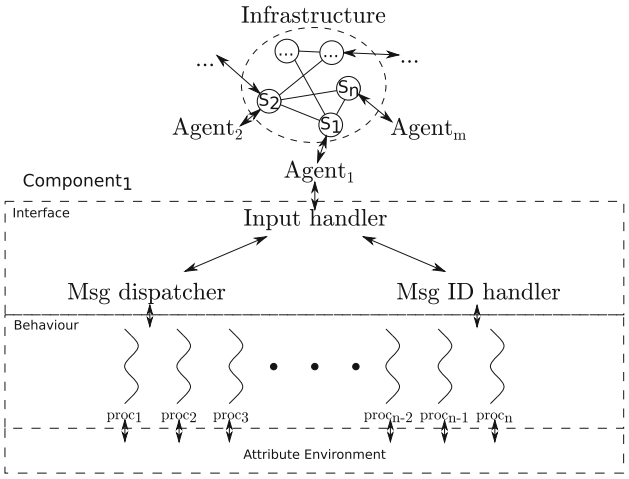


Fig. 2 A Component interface to a *GoAt* system

agent hides the details of the infrastructure from a component. An agent can be seen as a piece of software that handles the interaction between a component and the infrastructure server connected to it. A component registers to a server in the infrastructure by creating a dedicated agent through a registration address to handle their interactions.

In what follows, we provide a brief description of the implementation and of the dynamics of our distributed coordination infrastructures.

The Component. As reported in Fig. 2, a *GoAt* component consists of a behavioural part represented by its running processes and an interface (its agent) to deal with the infrastructure's server connected to it. The interface consists of three entities: the Input handler, the Msg dispatcher, and the Msg ID handler.

The *Input handler* is used to collect all incoming messages from the infrastructure's server and to forward reply messages to the Msg ID handler.

The *Msg dispatcher* stores a message in the waiting queue of the component until all messages with smaller id have been sent/delivered. Once this condition is satisfied, the Msg dispatcher forwards the message to a process; if the process accepts, the message is considered as delivered; otherwise, the Msg dispatcher tries with another process. The procedure continues until either the message is accepted by some process or all processes have rejected it. In both cases, the message is considered as delivered and the new id is notified to the Msg ID handler which updates the id of the next message to receive. It is important to note that any change to the attributes during the decision of accepting or rejecting the message can be only committed if the message is accepted; otherwise, it will be rolled-back.

The *Msg ID handler* deals with requests of processes wanting to send a message, and provides them with fresh ids. The handler forwards the request to the infrastructure's

server. While the process is waiting to send its message, dispatched messages are added to the waiting queue of the component. Once a reply message with a fresh id is received, the Msg ID handler forwards it to the process only when all messages with smaller id have been processed. The process can now manipulate the attributes environment and send a new message to the Msg ID handler which will forward it to the infrastructure's agent. All attribute updates are committed, and the msg dispatcher is notified about the new id.

The Coordination Infrastructures. These infrastructures are responsible for forwarding messages to components and also for issuing fresh message ids. Each kind of infrastructure consists of a set of server nodes that are logically connected in a specific way and collaborate to deliver sent messages to all connected components except for the sender. The implementation details of each infrastructure are reported below:

- *The Cluster Infrastructure* It consists of a registration node and a set of servers sharing a counter node and an input queue. A *GoAt* component needs to register itself to the cluster. The component contacts the registration node which will forward its network address to all cluster' servers. The component forwards its messages to the input queue of the cluster. A cluster server gets a message from the input queue which acts as a synchronisation point. If the message is a request for a fresh id, the server asks for a fresh id from the counter node and sends a reply back to the requester; otherwise, the message is forwarded to all components connected to the cluster except for the sender. This kind of infrastructure is a straightforward generalisation of a centralised implementation where only a single server is responsible for forwarding and sequencing messages.
- *The Ring Infrastructure* It consists of a registration node and a set of servers sharing a counter node. Upon registration, a component is only registered to one server (a parent) in the ring. This server will be the only interface for the component to interact with the infrastructure. The fact that components are assigned to specific servers allows us to reuse the same TCP connection. This would avoid unnecessary delays caused by re-establishing the connection every time a message is exchanged. A component forwards its messages to the input queue of its parent server. A ring server gets a message from its input queue: if it is a request message, the server asks for a fresh id from the counter node and sends a reply back to the requester; otherwise, the message is forwarded to all components directly connected to this server except for the sender. The message is also forwarded to its neighbour in the ring. When a server receives a message from its neighbour, it will accept the message only when its id is greater than the id of the last message processed at this server; otherwise, the message is discarded.
- *The Tree Infrastructure.* The tree infrastructure consists of a *registration node* and a set of servers organised in a logical tree. The *registration node* handles the construction of the infrastructure and the registration of components. When a component registers to the infrastructure through its agent, the registration node associates the agent to a specific server by assigning it communication ports to manage interaction with the selected server. The root of the tree is the only server that is responsible for generating sequence numbers. Each server is responsible for a group of agents and has its own input queue. The agent forwards its component messages to the input queue of its parent server. The server gets a message from its input queue: if it is a request message and the server is the root of the tree, the server assigns it a fresh id and sends a reply back to the requester; otherwise, the server forwards the message to its parent until the message reaches the root. Every time a request message traverses a server, it records its address in a linked list to help trace back the reply to the original requester with a minimal number of messages. If the server receives a reply message, it will forward it to the address on top of the message's linked list storing the path. As a consequence, this address is removed from the linked list. Finally, when a data message is received, it is forwarded to all connected agents and servers except for the sender.

In what follows, we briefly introduce the programming constructs of the *GoAt* API and show how they relate to the *AbC* primitives.

4.2 The programming interface

The main programming constructs of the *GoAt* API are reported in Fig. 3.

A component is the main building block of a *GoAt* system; each component contains a set of processes, defining its behaviour, and a set of attributes, defining its run-time status and contextual data. A *GoAt* system consists of a collection of *GoAt* components. Components execute in parallel and exchange messages only through message passing. In Fig. 3, Part 1, we show how to define a *GoAt* component, connect it to an infrastructure, and manipulate its attribute values. The method NewComponent(Agent,Environment) takes an agent which is created based on the registration address of an infrastructure Addr. It also takes an attribute environment Environment and creates a *GoAt* component. Components

Part 1: Initialization

```
1  Component:=goat.NewComponent(Agent,Environment)
2
3  Agent := goat.NewTreeAgent(Addr)
4
5  Environment := map[string]interface{}
6
7  Comp(Attribute)
8
9  Set(Attribute, Value)
```

Part 2: Behaviour

```
1  goat.NewProcess(Component).Run(proc *goat.Process)
```

Part 3: Process declaration

```
1  func(proc *goat.Process){
2    proc.Command_1
3    ...
4    proc.Command_n
5    }
```

Part 4: Commands

```
1  Send(Tuple, Predicate)
2
3  Receive(acceptFnc func(Attributes,Tuple) bool)
4
5  SendUpd(Tuple, Predicate, updFnc)
6
7  GSendUpd(Guard, Tuple, Predicate, updFnc)
8
9  Spawn(Process)
10
11 Call(Process)
12
13 WaitUntilTrue(Predicate)
14
15 Select(cases ...Case)
16
17 Case(Predicate, Action, Process)
```

Part 5: Predicates

```
1  Equals(_,_),   And(_,_),   Belong(_,_), Not(_), etc ...
```

Fig. 3 The *GoAt* API

are parametric with respect to the infrastructure that mediates their interactions and the programmer needs only to know the registration address of components in the infrastructure. Currently, three types of infrastructures are supported, namely Tree, Cluster, and Ring. The attribute environment of a component is defined as a map from attribute identifiers to their values. The attributes of a component can be retrieved and set via the methods Comp(attribute) and Set(attribute, value), respectively.

Example (step 6/7): In our example, we create a vertex as follows:

```
environment := map[string]interface{}{"round": 0, "used": {}, ...}
agent := goat.NewTreeAgent("127.0.0.1:17000")//registration address
vertex := goat.NewComponent(agent, environment)
```

In Fig. 3, Part 2, the method Run is used to assign a behaviour to a *GoAt* component and also to start its execution. This method takes a process as an argument and executes it within the scope of the current component. The code inside the Run method represents the actual behaviour of a component.

Example (step 7/7): In our example processes, processF, processT, processD and processA are created inside the vertex and they start executing as follows:

```
1  goat.NewProcess(vertex).Run(
2          func(proc *goat.Process) {
3              proc.Spawn(processF)
4              proc.Spawn(processT)
5              proc.Spawn(processD)
6              proc.Spawn(processA)
7           }
8          )
```

Processes processF, processT, processD, and processA code will be detailed in Sect. 4.3.

The generic behaviour of a *GoAt* process is implemented via a Go function as reported in Fig. 3, Part 3. This function takes a reference to a *GoAt* process and executes its commands. Note that beside *GoAt* commands, which will be explained later, the usual loop and branching statements of Go can also be used. Furthermore, in Fig. 3, Part 4, we define the available *GoAt* commands. The main communication actions, send and receive, are implemented via Send(Tuple, Predicate) and Receive (acceptfunc(attr ∗ Attributes, msgTuple)bool) methods. The send method communicates a tuple of values, Tuple, to components whose attributes satisfy the predicate Predicate. The receive method accepts a message and passes it to a Boolean function that checks if it satisfies the receiving predicate of a component. We also provide two other versions of the send action: a side-effect send SendUpd and a guarded side-effect send GSendUpd. The former has immediate attribute updates once executed, and the latter can also be guarded by a predicate Guard that blocks the execution until the guard is satisfied.

Spawn dynamically creates a new process Process and executes them in parallel with the main process at run-time, while Call(Process) implements a process call. The awareness operator, implemented via the method WaitUntilTrue (Predicate), blocks the execution of a process until predicate Predicate is satisfied. The method Select(cases ...Case) is a non-deterministic selection of guarded processes. This method takes a finite number of arguments of type Case, each of which is composed of an action guarded by a

predicate and a continuation process as shown in the syntax of a case. When the guarding predicate of one branch is satisfied, the method enables it and terminates other branches. Finally, Part 5 shows some of the supported predicates, i.e. Equals, And, Belong, and Not correspond to $=$, $\wedge$, $\in$ and $\neg$, respectively. Other standard predicates are also available. We use `Receiver(a)` in the sender predicate to evaluate the predicate based on the value of attribute `a` in the receiver side. For instance, `Belong( goat.Comp("id") , goat.Receiver("N"))` is equivalent to `this`.id $\in$ N.

4.3 Case study: a distributed graph colouring

In this section, we show how to use the programming constructs of the *GoAt* API to program a distributed variant of the graph colouring algorithm [24] presented in the running example. We render the problem as a typical CAS scenario where a collective of agents, executing the same code, collaborate to achieve a system-level goal without any centralised control. To avoid verbosity, we omit all auxiliary functions, but we comment on their behaviour.

Process F, reported below, proposes a colour. If a vertex is not assigned a colour and the value of attribute send_try is true, the process sends a try message to its neighbours identifying them by the predicate Belong(goat.Comp("id"), goat.Receiver("N")). The try message contains a try label, the proposed colour, the current round, and the id of this vertex. The proposed colour is the smallest colour that has not yet been selected by neighbours (not in used). The function Evaluate(minColorNot, goat.Comp("used")) is used to propose a colour. As side effects, the attribute colour is assigned the new colour and the attribute send_try is set to false.

```
1  func processF(proc *goat.Process) {
2    for {
3      proc.GSendUpd(goat.And(goat.Equals(goat.Comp("assigned"), false),
         goat.Equals(goat.Comp("send_try"), true)), goat.NewTuple("try",
         goat.Evaluate(minColorNot, goat.Comp("used")),
         goat.Comp("round"), goat.Comp("id")),
         goat.Belong(goat.Comp("id"), goat.Receiver("N")),
4        func(attr *goat.Attributes){
5          attr.Set("colour", minColorNot(attr.GetValue("used")))
6          attr.Set("send_try", false) })
7    }}
```

Process T deals with try-messages ("try", y, z, tid) as mentioned before. If the current round equals the round attached in the message z, then the received message has been originated by another component performing the same round of the algorithm and we have two cases (Lines 12-19). The first case is executed when the id of the vertex is greater than the id of the message tid, i.e. the sender has an id smaller than the id of the receiver. In this case, the message is ignored (there is no conflict), simply the counter of received messages is incremented. In the second case, the received colour is recorded to check the presence of conflicts. The value of y is added to constraints, and the counter is incremented by 1. If z is greater than the current round, as in (Lines 21-32), then the received message has been originated by a component executing a successive round and two possible alternatives are considered (thisId $>$ tid or thisId $<$ tid). In both cases, round is set to z, send_try and counter are updated accordingly, and constraints is set to the value of y if thisId $<$ tid.

```
1  func processT(proc *goat.Process) {
2    for {
3      proc.Receive(func(attr *goat.Attributes, msg goat.Tuple) bool{
4        if msg.IsLong(4) && msg.Get(0) == "try" {
5          y := msg.Get(1)
6          z := msg.Get(2).(int)
7          tid := msg.Get(3).(int)
8
9          thisRound := attr.GetValue("round").(int)
10         thisId := attr.GetValue("id").(int)
11
12         if thisRound == z {
13           if thisId > tid {
14             attr.Set("counter", attr.GetValue("counter").(int) + 1)
15             return true
16           } else if thisId < tid {
17             attr.Set("counter", attr.GetValue("counter").(int) + 1)
18             attr.Set("constraints", add(attr.GetValue("constraints"), y))
19             return true
20           }
21         } else if thisRound < z {
22           if thisId > tid {
23             attr.Set("round", z)
24             attr.Set("send_try", true)
25             attr.Set("counter", 1)
26             attr.Set("constraints", goat.NewTuple())
27             return true
28           } else if thisId < tid {
29             attr.Set("round", z)
30             attr.Set("send_try", true)
31             attr.Set("counter", 1)
32             attr.Set("constraints", goat.NewTuple(y))
33             return true
34           }
35         }
36       }
37       return false
38     })}
```

Process D, below, is used to receive done-messages of the form ("done", y, z, tid) where y is the assigned colour, z is the attached round, and tid is the sender id. These are sent by components that have reached a final decision about their colour. We have two cases: either the attribute round is $<$ z or $\geq$ z. In both cases, the used colour is registered in used and the counter done is incremented. However, in the second case, private attributes are updated to indicate the startup of a new round (z).

```
1  func processD(proc *goat.Process) {
2    for {
3      proc.Receive(func(attr *goat.Attributes, msg goat.Tuple) bool{
4        if msg.IsLong(4) && msg.Get(0) == "done" {
5          if attr.GetValue("round").(int) < msg.Get(2).(int) {
6            attr.Set("round", msg.Get(2))
7            attr.Set("constraints", goat.NewTuple())
8            attr.Set("send_try", true)
9            attr.Set("counter", 0)
10         }
11         attr.Set("done", attr.GetValue("done").(int) + 1)
12         attr.Set("used", add(attr.GetValue("used"), msg.Get(1)))
13         return true
```

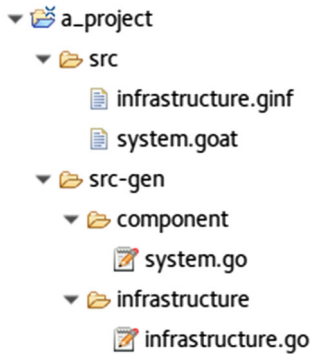


Fig. 4 The *GoAt* plugin

```
14          } else {
15            return false
16          }}}}}
```

Process A, reported below, is used to assign a final colour to a vertex. It can only be executed when messages from neighbours (which are not assigned colours) have been received and no conflict has been found (i.e. the colour is neither in used nor in constraints). When the above conditions are satisfied, message (“done",
colour, round + 1, id) is sent to neighbours, the attribute Assigned is set to true, and the process terminates.

```
1  func processA(proc *goat.Process) {
2    proc.GSendUpd(goat.Equals(goat.Evaluate(canAssign,
         goat.Comp("counter"), goat.Comp("N"), goat.Comp("done"),
         goat.Comp("colour"), goat.Comp("constraints"),
         goat.Comp("used")), true), goat.NewTuple("done",
         goat.Comp("colour"), goat.Evaluate(inc, goat.Comp("round")),
         goat.Comp("id")), goat.Belong(goat.Comp("id"), goat.Receiver("N")),
3      func(attr *goat.Attributes){attr.Set("assigned", true)})
4  }
```

4.4 The eclipse plugin for *GoAt*

In this section, we would like to briefly comment on the Eclipse plugin[2] we have developed for *GoAt*. The main goal of the *GoAt* plugin is to permit programming in a high-level syntax (i.e. the syntax of the original calculus *AbC*).

This syntax can be then analysed via formal methods by relying on the operational semantics of the *AbC* calculus. Once the code has been analysed, the *GoAt* plugin will generate formally verifiable Go code because of the correspondence results we prove in this article. In this article, we focus on the implementation part and we will consider verification tools for future works.

Figure 4 shows the project explorer of a *GoAt* plugin project. The source folder `src` consists of two main files: the infrastructure file with `.ginf` extension and the system file with `.goat` extension. The infrastructure file is used to create an infrastructure which can be of three types: cluster, ring, and tree. We also support local concurrency. The system file contains the actual *GoAt* specifications and a reference to the infrastructure that mediates the interaction between *GoAt* components. Once these files are saved, the *GoAt* plugin automatically generates Go code in the `src-gen` folder. We plan to integrate, in the near future, formal tools and rely on static analysis to inspect *GoAt* specifications before code generation.

Below, we show how Process F in Sect. 4.3 would be written using the Eclipse plugin. Clearly, the syntax is clean and less verbose which helps the modellers to focus on the problem they are trying to solve rather than worrying about complicated syntactic constructions.

```
process F {
  loop{
    if(comp.send_try && !comp.assigned)
    send{"try", minFeasibleColor(comp.used), comp.round,
       comp.id}@(comp.id in receiver.N)[ comp.send_try := false,
                comp.colour := minFeasibleColor(comp.used)];
  }}
```

Other examples can be found in the WebPage of *GoAt*. There, we also show how to program a complex and sophisticated variant of the well-known problem of Stable Allocation in Content Delivery Network (CDN) [27] using the *GoAt* plugin. We show that although our solution is more open and less restrictive, the complexity of our solution is still comparable to the original one adopted by Akamai’s CDN, one of the largest distributed systems available.

5 Performance evaluation

We compare the above-mentioned infrastructures by modelling them in terms of a Continuous Time Markov Process [32]. Note that we use Markov Processes to model and evaluate our infrastructures because it is very hard, or even impossible, to perform statistical analyses of a *real* distributed system when the number of participants is large, like in the scenarios considered in this section. We do not have enough computation resources to set up large infrastructures and conduct the comparison. If we rely on small infrastructures, we would get subjective results. It is clear that the tree would perform better in large settings, while the cluster performs better in small ones. Thus, all results in this section are based on stochastic simulation which anyway provides a quality approximation of the actual performance of the *GoAt* implementation.

The state of a process represents possible infrastructure configurations, while the transitions (that are selected probabilistically) are associated with events on messages. We have three types of events: a new message *sent* by a component; a message *transmitted* from a node to another in the infrastructure; a message locally *handled* by a node (i.e. removed from an input/waiting queue). Each event is associated with

[2] https://github.com/giulio-garbi/goat-plugin.

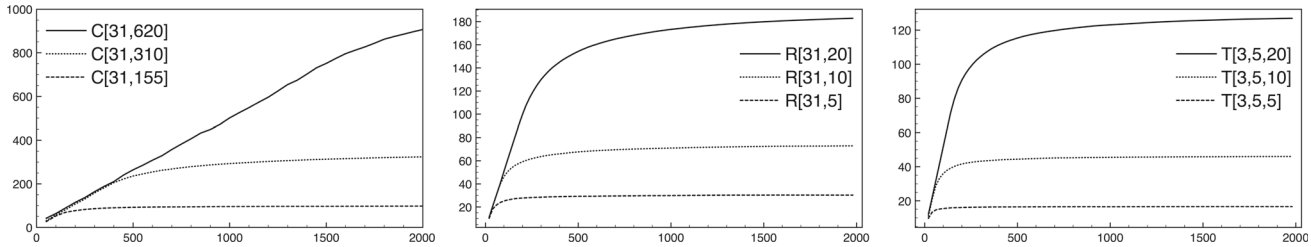


Fig. 5 DP scenario: x-axis: simulation time and y-axis: avg. delivery time for cluster, ring, and tree

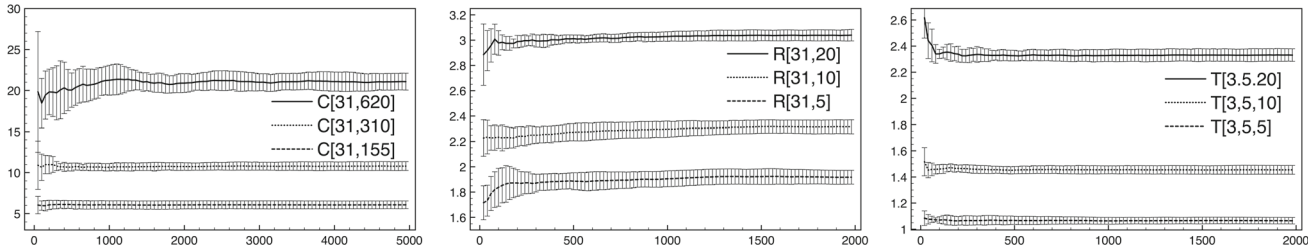


Fig. 6 DP scenario: x-axis: simulation time and y-axis: avg. message time gap for cluster, ring, and tree

a *rate* that is the parameter of the *exponentially distributed* random variable governing the *event duration*. We developed a simulator[3] for performance evaluation.

To perform the simulation, we need to fix three parameters: the *component sending rate* λ_s; the *infrastructure transmission rate* λ_t; and the *handling rate* λ_h. In all experiments, we fix the following values: $\lambda_s = 1.0$, $\lambda_t = 15.0$, and $\lambda_h = 1000.0$ and rely on kinetic Monte Carlo simulation [33]. The infrastructure configurations are defined as follows:

- $C[x, y]$, indicates a *cluster* with x nodes and y components;
- $R[x, y]$ indicates a *ring* with x nodes each of which manages y components;
- $T[x, y, z]$ indicates a *tree* with x levels. Each node (but the leafs) has $y + z$ children: y nodes and z components. A leaf node has z components.

We consider two scenarios: (1) *Data Providers* (DP): In this scenario, only a fraction of components send data messages that they, for example, acquire via sensors in the environment where they operate. An example could be a *Traffic Control System* where *data providers* are devices located in the city and *data receivers* are the vehicles travelling in the area; (2) *communication intensive* (CI): This scenario is used to estimate the performance when all components send messages continuously at a fixed rate so that we can evaluate situations of overloaded infrastructures. The former scenario is more realistic for CAS.

We consider two measures: the average delivery time and the average message time gap. The first measure indicates the time needed for a message to reach all components, while the latter indicates the interval between two different messages received by a single component. Data provider scenario (DP)

We consider configurations with 31 server nodes 155, 310, or 620 components and assume that only 10% of the components is sending data. The average delivery time is reported in Fig. 5, while the average message time gap (with confidence intervals) is reported in Fig. 6. The tree structure offers the best performance, while the cluster one is the worst. When the cluster reaches an equilibrium (at time $\sim$ 2000), $\sim$ 90 time units are needed to deliver a message to 155 components, while the ring and the tree need only $\sim$ 25 and $\sim$ 10 time units, respectively. The reason is that in the cluster all server nodes share the same input queue, while in the tree and the ring each server node has its own queue. We can also observe that the performance of the ring in this scenario is close to the one of the tree. Moreover, in the cluster, the performance degrades when the number of components increases. This does not happen for the tree and the ring. Finally, we can observe that messages are delivered more frequently in the ring ($\sim$ 1.9 time units) and the tree ($\sim$ 1.1 time units) than in the cluster ($\sim$ 5.5 time units) as reported in Fig. 6.
Communication intensive scenario (CI) We consider infrastructures composed by 155 components that continuously send messages to all the others. Simulations are based on the following configurations:

- Cluster-based: $C[10, 155]$, $C[20, 155]$ and $C[31, 155]$;
- Ring-based: $R[5, 31]$ and $R[31, 5]$;
- Tree-based: $T[5, 2, 5]$ and $T[3, 5, 5]$.

[3] https://bitbucket.org/Lazkany/abcsimulator.

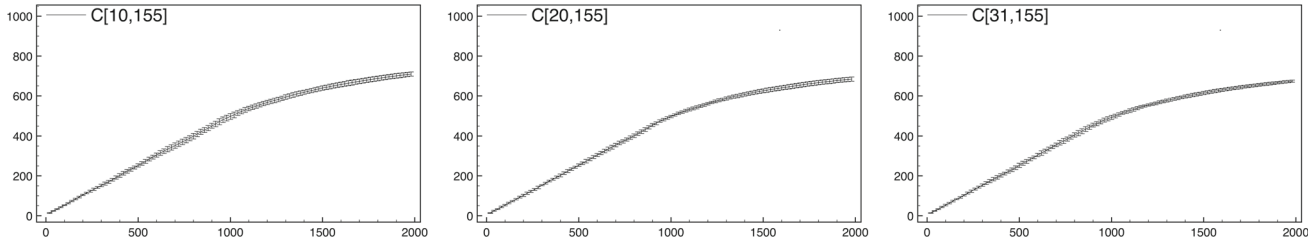


Fig. 7 CI scenario: x-axis: simulation time and y-axis: avg. delivery time for cluster with 10/20/31 servers

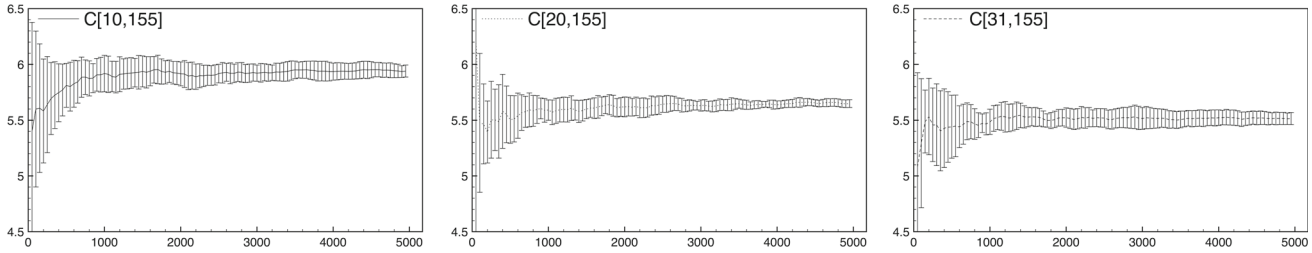


Fig. 8 CI scenario: x-axis: simulation time and y-axis: avg. message time gap for cluster with 10/20/31 servers

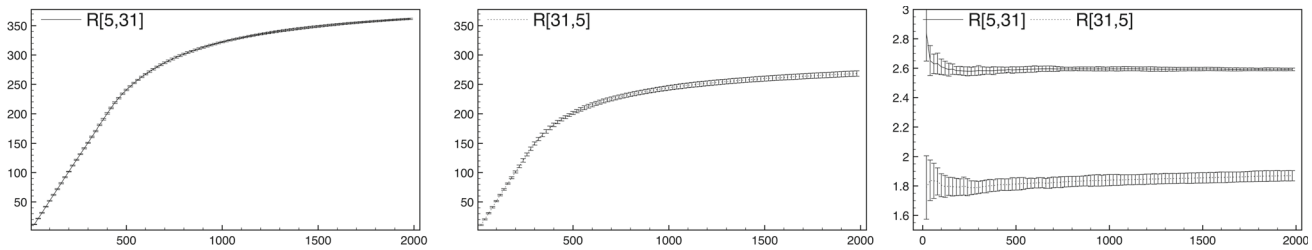


Fig. 9 CI scenario: x-axis: simulation time and y-axis: avg. delivery time and avg. message time gap for ring with 5 and 31 servers

Figure 7 shows the average delivery time for a cluster of 155 connected components and 10, 20 or 31 server nodes.Clearly, when the cluster reaches an equilibrium ($\sim$ 2000), $\sim$ 800 time units are needed to deliver a message to all components. We also observe that the number of server nodes in the cluster has a minimal impact on this measure because they all share the same input queue. The *Average Message Time Gap* of the cluster, in Fig. 8, indicates that in the long run a component receives a message every 6/5.5 time units even if the number of servers is increased from 10 to 31.

Better performance can be obtained for the ring infrastructure. In the first two plots of Fig. 9, we report the average delivery time for the configurations $R[5, 31]$ and $R[31, 5]$. The last plot compares the average message time gap of the two configurations. In the first one, a message is delivered to all the components in 350 time units, while in the second one 250 time units are needed. This indicates that increasing the number of nodes in the ring enhances performance. This is because in the ring all nodes cooperate to deliver a given message. Also the time gap decreases, i.e. a message is received every 2.6 and 1.8 time units.

Figure 10 shows how the *average delivery time* changes during the simulation for $T[5, 2, 5]$ and $T[3, 5, 5]$. The two configurations have exactly the same number of nodes (31) with a different arrangement. The two configurations work almost in the same way: a message is delivered to all the components in about 120 time units. Clearly, the tree is 5-time faster than the cluster and 2-time faster than the ring. Moreover, in the tree-based approach, a message is delivered to components every $\sim$ 1.1 time units as reported in Fig. 11. This means that messages are constantly delivered after an initial delay.

The summarised results of the infrastructures with 31 nodes when measuring the average delivery time for 155 and 310 connected components are reported in Fig. 12. These results show that tree infrastructures offer the best performance; cluster-based ones do not work well, while ring-based ones are in between the two. Differences become clearer when increasing the number of components from 155 to 310 (right side of Fig. 12).

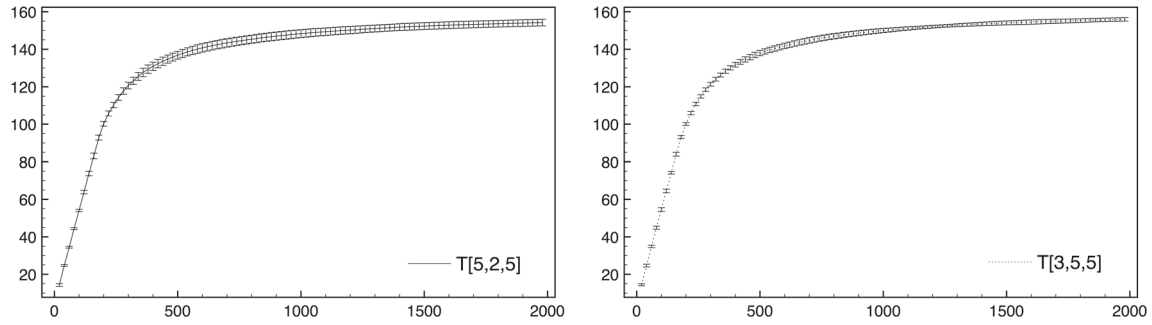


Fig. 10 CI scenario: x-axis: simulation time and y-axis: avg. delivery time: tree/ $T[5, 2, 5]$ and $T[3, 5, 5]$

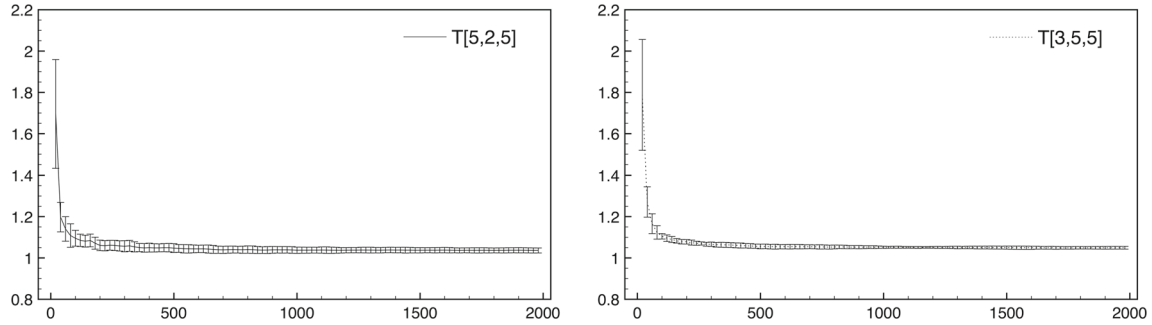


Fig. 11 CI scenario: x-axis: simulation time and y-axis: avg. message time gap: tree/ $T[5, 2, 5]$ and $T[3, 5, 5]$

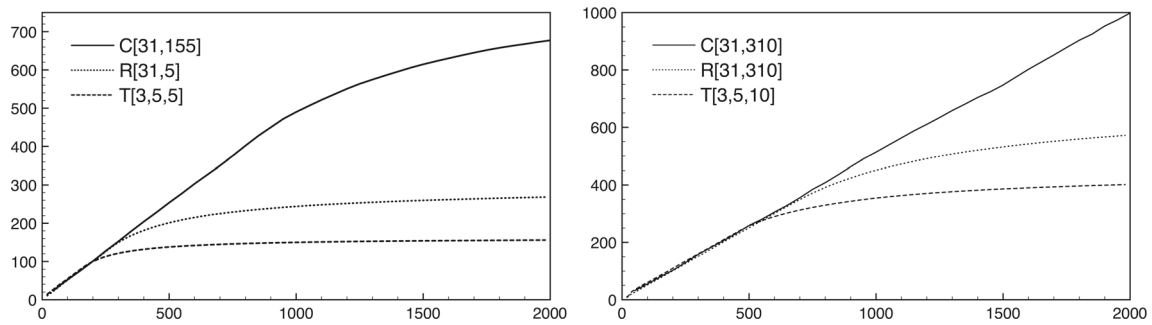


Fig. 12 Summary: x-axis: simulation time and y-axis: avg. delivery time cluster/ring/tree (155/310 comp.)

6 Related work

In this section, we relate our work to existing ones in terms of: collective formation, adaptation and distributed coordination through total-order broadcast.

Several frameworks have been proposed to target the problem of collective (or ensemble) formation. These approaches usually differ in the way they represent collectives and in their generality. The SCEL language [19], proposed in the ASCENS project [35], is a kernel language that has been designed to support the programming of autonomic computing systems. This language relies on the notions of *autonomic components* representing the collective members, and *autonomic component ensembles* representing collectives. Each component is equipped with an interface, consisting of a collection of attributes, describing its features. Attributes are used by components to dynamically organise themselves into ensembles and as a way to select partners for interaction. SCEL has inspired the development of the core calculus *AbC* to study the impact of attribute-based communication. Compared with SCEL, the knowledge representation in *AbC* is abstract and is not designed for detailed reasoning during the model evolution. This reflects the different objectives of SCEL and *AbC*. While SCEL focuses on programming issues, *AbC* concentrates on a minimal set of primitives to study attribute-based communication.

DEECo [14] and Helena [25] adopt an architecture-based approach to specify ensembles. In general, they impose logical conditions on the membership of ensembles that components have to satisfy. As opposed to $\mathcal{GoAt}$, the ensemble is a first class entity in these languages, while in $\mathcal{GoAt}$ ensemble

are logical entities implied by the run-time status of components and the interaction predicates.

JULIA [13] is also architecture based and achieves adaptation by defining systems that can adapt their configurations with respect to contextual conditions. System components manipulate their internal structure by adding, removing, or modifying connectors. However, in this approach interaction is still based on explicit connectors. In $\mathcal{GoAt}$ predefined connections simply do not exist, and we do not assume a specific architecture or containment relations between components.

Furthermore, Context Oriented Programming [22] provides linguistic primitives is to define context-dependent behavioural variations. These variations are expressed as partial definitions of modules that can be overridden at run-time to adapt to contextual information. They can be grouped via layers to be activated or deactivated together dynamically. These layers can be also composed according to some scoping constructs. Our approach is different in that components adapt their behaviour by considering the run-time changes of the values of their attributes which might be triggered by either contextual conditions or by local interaction.

Other approaches that target similar problems revolve around team formation [30]. However, these approaches are usually developed for a specific purpose and thus have limited reusability. A more general and reusable framework, named RMASBench [26], was proposed for multi-agent coordination. Its main focus is on benchmarking of distributed constraints optimisation algorithms used within this framework to resolve team formation of agents.

Regarding total-order broadcast protocols, we would like to mention (1) the fixed sequencer approach [18], (2) the moving sequencer approach [16], and (3) the privilege-based approach [17]. The first approach is centralised and relies on a single sequencer of messages. We can consider the cluster infrastructure as a natural generalisation of this approach where instead of a single server, many servers collaborate to deliver messages. The second approach is similar to the ring infrastructure with the only exception that the role of the sequencer is transferred between the ring servers. This is achieved by circulating a specific token between ring servers. However, the liveness of this approach depends on the token and fairness is hard to achieve if one server has a larger number of senders than the other servers. The third approach relies on consensus between components to establish a total order. As mentioned before, consensus-based approaches are not suitable for open systems and they cannot deal with component failures.

7 Concluding remarks and future works

We proposed a distributed coordination infrastructure for the *AbC* calculus, and we proved its correctness. We developed a corresponding programming API, named $\mathcal{GoAt}$, to exploit the main interaction primitives of the *AbC* calculus directly in Go. The actual implementation of the API fully relies on the formal semantics of *AbC* and is parametric with respect to the coordination infrastructure that manages the interaction between components. We used the $\mathcal{GoAt}$ API to program a distributed variant of the graph colouring problem and commented about the simplicity of its use. We evaluated the performance of the proposed infrastructure with respect to others. The results showed that our infrastructure exhibits a good performance. We also developed an Eclipse plugin for $\mathcal{GoAt}$ to permit programming in a high-level syntax which is less verbose and helps programmers to focus on the problem they want to solve rather than worrying about complicated syntactic constructions.

We consider the tools that we have developed so far as a starting point for integrating formal tools that analyse the $\mathcal{GoAt}$ plugin code and ensure that it satisfies specific properties before code generation. We also plan to enhance the implementation of our infrastructures by considering fairness and reliability issues. We also want to consider the challenging problem of verifying collective properties of $\mathcal{GoAt}$ code. As a step in this direction, we developed the ReCiPe framework [10] (a symbolic representation of *AbC* specifications) and we extended LTL to be able to specify collective properties.

Acknowledgements Open access funding provided by University of Gothenburg.

References

1. Abd Alrahman, Y., De Nicola, R., Loreti, M.: On the power of attribute-based communication. In: Formal Techniques for Distributed Objects, Components, and Systems—36th IFIP International Conference, FORTE. Springer, pp. 1–18 (2016)
2. Abd Alrahman, Y., De Nicola, R., Loreti, M.: Programming of CAS systems by relying on attribute-based communication. In: Leveraging Applications of Formal Methods, Verification and Validation: Foundational Techniques—7th International Symposium, ISoLA '16, Proceedings, Part I. Springer, pp. 539–553 (2016)

3. Abd Alrahman, Y., De Nicola, R., Loreti, M.: Programming the Interactions of Collective Adaptive Systems by Relying on Attribute-based Communication. ArXiv e-prints (2017)
4. Abd Alrahman, Y., De Nicola, R., Loreti, M., Tiezzi, F., Vigo, R.: A calculus for attribute-based communication. In: Proceedings of the 30th Annual ACM Symposium on Applied Computing, SAC '15. ACM, pp. 1840–1845 (2015)
5. Agha, G.: Actors: A Model of Concurrent Computation in Distributed Systems. MIT Press, Cambridge (1986)
6. Alrahman, Y.A., De Nicola, R., Garbi, G.: GoAt: Attribute-based interaction in Google go. In: Leveraging Applications of Formal Methods, Verification and Validation. Distributed Systems—8th International Symposium, ISoLA 2018, pp. 288–303 (2018)
7. Alrahman, Y.A., De Nicola, R., Garbi, G., Loreti, M.: A distributed coordination infrastructure for attribute-based interaction. In: Formal Techniques for Distributed Objects, Components, and Systems—38th IFIP WG 6.1 International Conference, FORTE 2018, Proceedings, pp. 1–20 (2018)
8. Alrahman, Y.A., Mezzina, C.A., Vieira, H.T.: Testing for coordination fidelity. In: Boreale, M., Corradini, F., Loreti, M., Pugliese, R. (eds.) Models, Languages, and Tools for Concurrent and Distributed Programming, pp. 152–169. Springer, Berlin (2019)
9. Alrahman, Y.A., Nicola, R.D., Loreti, M.: A calculus for collective-adaptive systems and its behavioural theory. Inf. Comput. **268**, 104457 (2019)
10. Alrahman, Y.A., Perelli, G., Piterman, N.: A computational framework for adaptive systems and its verification. CoRR (2019). arXiv:1906.10793
11. Alrahman, Y.A., Vieira, H.T.: A coordination protocol language for power grid operation control. J. Log. Algebraic Methods Program. **109**, 100487 (2019)
12. Bonabeau, E.: From classical models of morphogenesis to agent-based models of pattern formation. Artif. Life **3**(3), 191–211 (1997)
13. Bruneton, E., Coupaye, T., Leclercq, M., Quéma, V., Stefani, J.-B.: The fractal component model and its support in Java. Softw. Pract. Exp. **36**(11–12), 1257–1284 (2006)
14. Bures, T., Plasil, F., Kit, M., Tuma, P., Hoch, N.: Software abstractions for component interaction in the internet of things. IEEE Comput. **49**(12), 50–59 (2016)
15. Camazine, S., Deneubourg, J., Franks, N.R., Sneyd, J., Theraulaz, G.: Self-organization in Biological Systems. Princeton studies in complexity. Princeton University Press, Princeton (2003)
16. Chang, J.-M., Maxemchuk, N.F.: Reliable broadcast protocols. ACM Trans. Comput. Syst. **2**, 251–273 (1984)
17. Cristian, F.: Asynchronous atomic broadcast. IBM Tech. Discl. Bull. **33**(9), 115–116 (1991)
18. Cristian, F., Mishra, S.: The pinwheel asynchronous atomic broadcast protocols. In: Second International Symposium on Autonomous Decentralized Systems, 1995. Proceedings. ISADS 95, pp. 215–221. IEEE (1995)
19. De Nicola, R., Latella, D., Lluch-Lafuente, A., Loreti, M., Margheri, A., Massink, M., Morichetta, A., Pugliese, R., Tiezzi, F., Vandin, A.: The SCEL language: design, implementation, verification. In: Wirsing, M., Hölzl, M., Koch, N., Mayer, P. (eds.) Software Engineering for Collective Autonomic Systems-The ASCENS Approach, pp. 3–71. Springer, Berlin (2015)
20. Ferscha, A.: Collective adaptive systems. In: Adjunct Proceedings of the 2015 ACM International Joint Conference on Pervasive and Ubiquitous Computing and Proceedings of the 2015 ACM International Symposium on Wearable Computers, UbiComp/ISWC'15 Adjunct, pp. 893–895, New York, NY, USA. ACM (2015)
21. Fischer, M.J., Lynch, N.A., Paterson, M.S.: Impossibility of distributed consensus with one faulty process. J. ACM **32**(2), 374–382 (1985)
22. Hirschfeld, R., Costanza, P., Nierstrasz, O.: Context-oriented programming. J. Object Technol. **7**(3), 125–151 (2008)
23. Hoare, C.A.R.: Communicating sequential processes. Commun. ACM **21**(8), 666–677 (1978)
24. Jensen, T.R., Toft, B.: Graph Coloring Problems, vol. 39. Wiley, New York (1995)
25. Klarl, A., Cichella, L., Hennicker, R.: From Helena ensemble specifications to executable code. In: Formal Aspects of Component Software—11th International Symposium, FACS 2014, pp. 183–190 (2014)
26. Kleiner, A., Farinelli, A., Ramchurn, S.D., Shi, B., Maffioletti, F., Reffato, R.: Rmasbench: benchmarking dynamic multi-agent coordination in urban search and rescue. In: International Conference on Autonomous Agents and Multi-Agent Systems, AAMAS '13, Saint Paul, MN, USA, May 6–10, 2013, pp. 1195–1196 (2013)
27. Maggs, B.M., Sitaraman, R.K.: Algorithmic nuggets in content delivery. SIGCOMM Comput. Commun. Rev. **45**(3), 52–66 (2015)
28. Milner, R., Parrow, J., Walker, D.: A calculus of mobile processes, II. Inf. Comput. **100**(1), 41–77 (1992)
29. Mosses, P.D.: Modular structural operational semantics. J. Log. Algebraic Program. **60–61**, 195–228 (2004)
30. Parker, J., Nunes, E., Godoy, J., Gini, M.L.: Exploiting spatial locality and heterogeneity of agents for search and rescue teamwork. J. Field Robot. **33**(7), 877–900 (2016)
31. Prasad, K.: A calculus of broadcasting systems. In: TAPSOFT'91, pp. 338–358. Springer (1991)
32. Robertson, J.B.: Continuous-time Markov chains (W. J. anderson). SIAM Rev. **36**(2), 316–317 (1994)
33. Schulze, T.P.: Efficient kinetic monte carlo simulation. J. Comput. Phys. **227**(4), 2455–2462 (2008)
34. Vukolić, M.: The quest for scalable blockchain fabric: proof-of-work vs. bft replication. In: Camenisch, J., Kesdoğan, D. (eds.) Open Problems in Network Security, pp. 112–125. Springer, Cham (2016)
35. Wirsing, M., Hölzl, M.M., Koch, N., Mayer, P. (eds.): Software Engineering for Collective Autonomic Systems—The ASCENS Approach. Lecture Notes in Computer Science, vol. 8998. Springer, Berlin (2015)